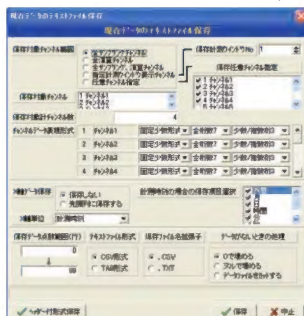
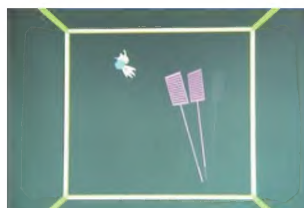
Interface

2004
11
Contents



[表紙デザイン：㈱プランニング・ロケッツ]





特集
現実の事象を回路やソフトウェアで処理するための必須の技術
技術者のためのデータ計測
Cover Story　Data measurement for engineers　59

Interface

2004 11

Contents

連載「フリーソフトウェア徹底活用講座」は,お休みさせていただきます.

日本最大のワイヤレス＆モバイル専門展示会

WIRELESS JAPAN 2004

北村 俊之

モバイルとワイヤレスの専門展示会「WIRELESS JAPAN 2004」(写真1)が，7月21日(水)～23日(金)の3日間，東京ビッグサイトで開催された．主催は(株)リックテレコム．今回で第9回を迎える同展示会は，ワイヤレス・ビジネスの拡大に向け，携帯電話キャリア，主要端末メーカ，無線LAN機器ベンダ，ワイヤレス・ソリューション・ベンダ，ソフトウェア・メーカ，アプリケーション・サービス・プロバイダ，通信機器ベンダなど約200社が出展し，前回を上回る規模での開催となった．また，初日の21日の午前中には，無線LAN製品の相互接続性認定団体「Wi-Fiアライアンス」が記者会見を行い，9月をめどに開始予定の新しい認定プログラムなどについての説明を行った．



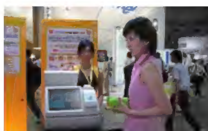
写真1　会場受け付け

さらに今回は，次世代のワイヤレス情報通信開発にフォーカスした「第1回 次世代ワイヤレス技術展」，モバイル端末の開発を支える部品にフォーカスした「モバイル電子部品フォーラム」，中堅中小企業経営のIT化にフォーカスした「中堅中小企業IT化経営展」の三つのイベントも同時に開催され，最終的な来場者数は，すべてのイベントを合わせて，32,836人にのぼった．今回は，「WIRELESS JAPAN 2004」および「第1回 次世代ワイヤレス技術展」を中心にレポートを行う．

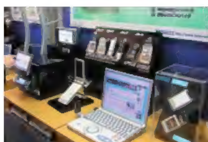
● WIRELESS JAPAN 2004

本展示会では，今回も携帯電話キャリアのブースや各メーカの最新機種やサービスに来場者の関心が集まっていた．

NTTドコモは，ウェアラブル・コンピュータ「UbiButton」や，非接触ICカード「FeliCa」を内蔵した「おサイフケータイ」(写真2)を中心にサービスの紹介を行っていた．「おサイフケータイ」では実際に，コンビニ(am/pm)のレジでの支払いや自動販売機での買い物など，携帯電話をかざして受けられる多様なサービスのデモを行っていた．このほか，カラオケでお気に入りの曲を簡単に呼び出せる機能や，ビル入館時に本人と確認する社員証の機能など，「FeliCa」の使用例を多数紹介していた．また同ブースでは，「FOMA対応 ビジュアルコントローラ」の展示デモが行われていた．こちらは，外出先のFOMAから「ビジュアルコントローラ」を介して，自宅のテレビやビデオ，内蔵カメラの映像を閲覧したり，エアコンや照明などの家電機器を遠隔操作することができるという(参考出品)．


写真2　NTTドコモのおサイフケータイ

DDIポケットは，データ通信サービス「Air H"」の最新端末の参考出品を行っていた(写真3)．こちらは，通信速度を現行(128kbps)の2倍の256kbpsとしており，料金定額で年末導入予定とのことだった．また，同ブースでは，SIMと呼ばれる小さなカードを入れ替えることで，同じ電話番号を他の端末でも使用可能にする，R-SIM (Radio-Subscribe Identify Model)方式を採用した，メール専用端末やGPS専用端末，ネット家電用リモコンなどを多数参考出展していた．状況に応じて腕時計やディジタル・オーディオ・プレーヤなどに，R-SIMカードに挿し込むことで，PHSの利用範囲が広がることを想定したソリューションである．

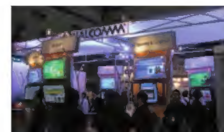

写真3　DDIポケットの「Air H"」最新端末

インテルのブース(写真4)では，ネットツーコムと富士通研究所が共同開発した「ユビキタスIP電話端末」が参考出展されていた．同製品は，無線LANと公衆無線網をシームレスに切り替えることが可能な携帯型無線IP電話端末で，2.2インチのQVGA液晶ディスプレイを搭載し，IEEE802.11bに対応している．OSは「Windows CE.NET 4.2」を搭載しており，Linuxへの対応も今後予定しているという．



写真4　インテルのブース

クアルコム(写真5)では，CDMAやBREWなどの携帯電話のチップや規格が多数展示されていた．今回来場者の高い関心を集めていたのは，KDDIのCDMA 1X WINにも採用されているEV-DOの次期バージョン，「EV-DO Rev.A」とのことだった．下り速度を2.4Mbpsから3.1Mbpsに，上りを144Kbpsから1.8Mbpsに高速化することで，IP電話やテレビ電話などネットワークの遅延が問題になるアプリケーションに対応することができるという．また，複数端末への同時配信(マルチキャスト)機能も追加しており，携帯電話でネットワーク・ゲームやテレビ会議が可能になるという．ただ，この技術がいつ頃導入されるかは，未定とのことだった．


写真5　クアルコムのブース

● 第1回 次世代ワイヤレス技術展

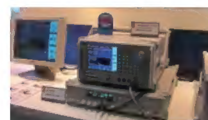
アドバンテストは，新製品である基地局(BS)用アンプ・モジュール評価に最適な「R3681 シングル・アナライザ」をはじめ，ネットワーク・アナライザ，スペクトラム・アナライザなど，同社の主力製品を一堂に展示していた．「R3681 シングル・アナライザ」(写真6)は，20Hz～32GHzという広い周波数測定範囲をもっており，高性能なスペクトラム解析と広帯域変調解析を1台で実現することができるという．


写真6　アドバンテストのR3681シングル・アナライザ

ローデ・シュワルツ・ジャパンは，スペクトラム・アナライザ，シグナル・ジェネレータなどの測定機器を多数展示していた．新製品でもあるハンドヘルド・スペクトラム・アナライザ「R&S FSH3」(写真7)は，100kHz～3GHzと広い周波数範囲をもち，小型・軽量(2.5kg，170×120×270mm)を実現している．100ファイルの波形，セットアップ保存をサポートし，各種通信規格の測定を可能にしている点も大きな特徴だという．


写真7　ローデ・シュワルツ・ジャパンのハンドヘルド・スペクトラム・アナライザ「R&S FSH3」

アンリツは，新製品である，高速・広帯域パワーメータ「ML2480Aシリーズ」，ディジタル放送信号アナライザ「MS8901A」(写真8)などを中心とした展示を行っていた．「ML2480Aシリーズ」は，各種ディジタル通信およびレーダ・システムのRFパワーを高精度に測定可能．50MHz～18GHzの測定周波数帯域を持ち，測定帯域幅は20MHz．TDMAおよびパルス波を，ゲーティングによるマルチ測定機能で，高精度に測定できるなどの特徴をもつとのことである．


写真8　アンリツのディジタル放送信号アナライザ「MS8901A」

日本テクトロニクス，バッテリ駆動でフローティング測定が可能なオシロスコープ「TPS2000シリーズ」を発売

■ 日時：2004年8月4日（水）
■ 場所：日本テクトロニクス（東京都港区）

日本テクトロニクス（株）は，バッテリ駆動で同時に4チャネルのフローティング測定が可能なフィールド向けディジタル・ストレージ・オシロスコープ「TPS2000シリーズ」を発売した．

同製品は，測定する4チャネルが独立しており，同一のグラウンドが取れないような環境での測定ができる．また，最大2Gsps/4チャネルでのディジタル・リアルタイム・サンプリングが可能であるほか，測定したデータをCFメモリ・カードに保存することもできる．8時間以上バッテリ駆動できるほか，バッテリを2個搭載しているため，電源を入れた状態でバッテリの入れ替えも可能．価格は￥389,000～．



TPS2014

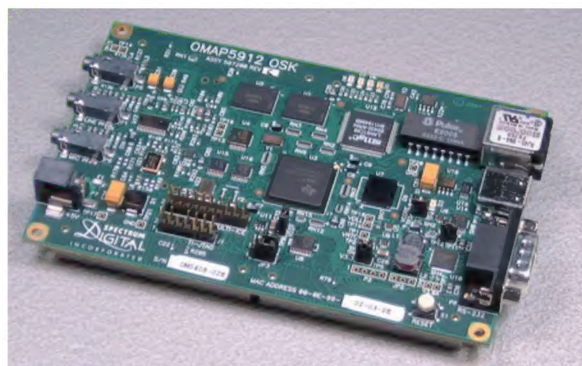
TI，OMAP Stater Kitを発売

■ URL：http://www.tij.co.jp/jsc/docs/dsps/product/omap/index.htm

日本テキサス・インスツルメンツ（株）は，ARM926（197MHz）とDSP TMS320C55xを1チップに内蔵したプロセッサ，OMAP 5912を搭載した学習キット「OMAP Starter Kit」を発売した．

同ボードにはEthernet，RS-232-C，USBが搭載されているほか，MontaVista製Linux Preview KitのCD-ROMが付属し，OMAPを使ったLinuxシステムの開発ができる．フラッシュ・メモリにはブート・ローダとカーネル，ファイル・システムが書き込まれた状態で出荷される．開発環境はRedHat Linux上に構築し，DSPプログラミングを行うための開発ツールも近く発表される．

価格は￥35,800．



OMAP Starter Kit

組み込み向けLinuxディストリビューションELinOS

■ URL：http://www.4link.co.jp/products/elinos.html

ドイツSYSGO社（http://www.sysgo.com/）から組み込み向けLinuxディストリビューションELinOSが発売された．国内ではFour Link Systems（http://www.4link.co.jp/）が同製品を取り扱う．

ELinOSはLinuxカーネル2.4.25，RTAI 3.0のサポート，Carrier Grade Linuxのほか，Eclipseベースの統合開発環境CODEOと，ターゲットの内部情報を表示するCOGNITOをもつ．COGNITOはイベント情報やプロファイル情報をグラフィカルに表示することができるツール．また，VxWorks Migration Kitなどもサポートし，他OSからの移行を支援する機能もある．



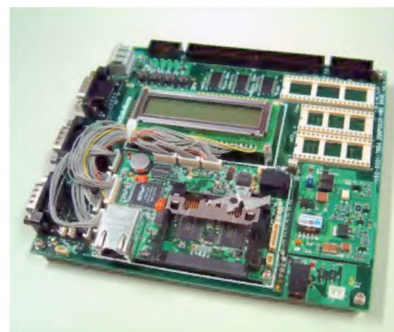
SYSGO CEO
Knut Degen氏

H8SでLinuxライクな環境を実現したeBoss-1

■ URL：http://www.cht.co.jp/chtcom/html/embe_00.html

コンピュータ・ハイテック（株）は，H8SマイコンでLinuxライクな環境を実現した「eBoss-1」を発売した．

同製品はH8Sマイコン搭載ボード「eBoss-1ボード」と独自開発のOS「eBoss-1カーネル」から構成される．eBoss-1カーネルはPOSIXに準拠し，ネットワーク機能を標準装備しているなど，「Linuxライク」な環境を提供している．そのほか，ROM/RAMともに256Kバイトあれば動作可能，ハード・リアルタイムのサポート，pthread準拠，数百msでの起動が可能などの特徴をもつ．eBoss-1評価キットのサンプル価格は￥98,000．



eBoss-1＋BASEボード

ハッカーの常識的見聞録



今月の常識

自動更新を今一度見直そう
――SUS 再び

広畑 由紀夫

☆ いよいよWindows XP SP2(以下SP2と略)の出荷とともに，パッチワークがより重要になってきました．今回は，ActiveDomainを使用しない環境でのローカル・ポリシの設定などから，パッチワークを見ていきます．

● SP2における弊害?!

SP2はセキュリティ面で非常に多くの変更がなされているため，ネットワーク向けセキュリティ製品のみならず，通常のアプリケーションにおいてもエラーが発生する可能性があります．それは，従来なら実行可能となっていたメモリ領域の一部が実行不可能になったり，初期設定の変更などがあったためです．

RC版においてすでにある程度の調査をし，現在対処中の読者も多いかもしれません．筆者もLAN向けのシステム・プログラムを多く組んでいるため，それらがSP2導入後も実行可能かどうかを検証する作業に追われています．

● 今後の対策

SP2に限らず，今後のことを考えれば，セキュリティの強化は必須といえるでしょう．しかし，セキュリティを強化した結果，それによって支障をきたすような場合も出てくることでしょう．さらには，修正プログラムの導入についても管理者が決定したいという場合ももちろん存在することでしょう．

今回は以前にも紹介しましたが，Software Update Services Server（SUS）1.0 with SP1を取り上げ，ActiveDomain以外でのローカル・ポリシについて触れてみます．

● SUSとSystem Management Server(SMS)

SMSは企業向けソフトウェア・マネージメントを管理するサーバウェアですが，アップデートの運用のみに使うには，いささか高価です．そこで，無償提供されているSUSによって導入とパッチ管理を行うという方法がアップデート対策だけを行う場合には有効でしょう．また，SUSは，Windows 2000においても動作するため，すぐに試験運用を行えるという点も魅力です．

では，SUSを使うメリットはどこにあるのでしょうか？ LAN上のローカル・サーバとして一元化管理することで，インターネット回線負荷の軽減効果はもちろんあるでしょう．しかし，今回はこの通信回線負荷の軽減よりも，「Windows XP SP2をあてさせないで自動更新を続けるクライアントを存続させるためにSUSを使用すること」が可能な点に着目します．

SUSでは，クライアントの自動更新に関して管理者の許可が必要になっています．過去のアップデート更新に関しては自動配布も可能になっていますが，新規もアップデータに関しては管理者が許可しなければクライアントに配布されません．そのことを利用して，SP2の導入を管理者が決められるわけです．

● SUSをActiveDomain以外のクライアント端末で簡単に使用する

SUSやSMSは，ActiveDomainで管理されているクライアント環境を前提にシステム管理を行うように設計されているようで，ActiveDomain外のクライアントに関する設定はなかなかめんどうです．さいわい，マイクロソフト社から情報が公開されているため，「グループ ポリシーまたはレジストリの設定を使用して自動更新を構成する方法」のシステム・レジストリ変更方法を参照してレジストリ情報ファイルを作成し，各クライアントに導入することでActiveDomainサーバの新規導入などを避けることができます．

筆者の場合もこの情報を元に，レジストリ情報ファイルを生成するアプリケーションを作成したうえで各端末のポリシを変更し，SUSにて複数の端末の自動更新を管理するように再設計しました．

● 今後のパッチワーク

以前よりSUS2.0が発表されていますが，SP2の遅れとともにSUS2.0も出荷が遅れているようです．無償のソフトウェアですがSUSはなかなか使い勝手のよい自動更新管理ツールなので，早く新バージョンを出してほしいものです．

現状，SUS2.0の配布が開始されるまでは，SUS1.0 with SP1で管理者がアップデートする時期やアップデート内容の把握と許可を行うのが，コスト的に安く済むと思われます．SP2への大規模なアップデートが行われるこの時期，今一度，自動更新にかかわるパッチワークを見直してみるべきでしょう．

- グループ ポリシーまたはレジストリの設定を使用して自動更新を構成する方法
  http://support.microsoft.com/default.aspx?scid=kb;ja;328010
- SUS1.0 with SP1配布サイト
  http://www.microsoft.com/downloads/details.aspx?familyid=a7aa96e4-6e41-4f54-972c-ae66a4e4bf6c&displaylang=ja
- マイクロソフト社によるSP2導入先延ばしツール(2004年8月16日から120日間有効)
  http://www.microsoft.com/technet/prodtechnol/winxppro/maintain/sp2aumng.mspx

ひろはた・ゆきお OpenLab.

移り気な情報工学

持続型技術——サスティナブル・テクノロジ

山本 強

社会学の分野にSustainable Society，持続型社会という用語がある．地球という閉鎖システムの中にいる以上，無限に経済成長を続けることはできないのは明らかであり，どこかでゼロ成長の安定状態にならなければならない．そういう状態に入った社会が持続型社会であり，欧州の先進国はすでにそういう状況にある，というものだ．

この段階に入ると経済成長率やGDPという成長期の指標では豊かさを測ることが難しくなる．すでに持続型社会の領域に突入した国々が，経済指標では明らかにアジア諸国に劣るにも関わらず，生活や文化で豊かさを感じるのは，彼らがすでに持続型社会にあるため，既存の「豊かさを測る指標」が適用できないという見方ができる．

競争型技術から持続型技術へのパラダイム・シフト

IT分野の技術開発は成長型，競争型の評価軸が目下の主流である．ネットワークは速度競争だし，デジカメも画素数競争になっている．しかし，こういった性能競争はいずれ限界点に行き着く．デジカメの画素数も30万画素，100万画素と上ってきたが，300万画素を超えたあたりから，スナップ写真用としてはオーバスペック気味になってきて，評価軸が価格やデザインという本来の性能ではないところに移っていった．性能が飽和し，デザインやユーザビリティも飽和した後の評価軸には何が残るのだろうか？ その先のキーワードとしてSustainability——持続可能性を考えてみたい．

環境分野にLCA——Life Cycle Assessmentという概念がある．ある工業製品を製造し，使用を終えて最終的に処分するまでにかかる全コストの評価値であり，それを環境負荷の指標とするわけである．しかし，環境にやさしいことだけを至上命題にする環境原理主義に陥いるだけでは芸がない．より快適な生活環境を低コスト，低環境負荷で維持することが，成長の限界に達した国や地域に求められる技術なのではないだろうか．そういった技術を総称してSustainable Technologyと呼ぶことにしたい．

持続の条件：メンテナンス性

性能が高いシステムは，性能を維持するためのコストも高いのが普通である．IT分野を見ても，10年前まではいわゆるメインフレーム・コンピュータなるものがあった．これが主流の座を降りたのは，UNIXワークステーションやPCなどと比べて，導入時の性能対コスト比が悪かったからである．システムを稼動状態に維持するコストはメインフレームもPCもあまり変わらないという話もある．低コストで高品質なメンテナンスを提供するのは，実は日本のお家芸でもある．100年以上前の木造家屋が普通に使われているというのは究極のメンテナンス技術があってのことといえる．

しかし，IT分野について見れば市場でアピールするのは性能指数や価格であり，メンテナンスに対する市場評価は決して高くない．価格は新興アジア地域との競争で，性能は欧米先進国との競争で，それに勝つのが当面の課題である．しかし，日本が本当に強いのはメンテナンスの技術なのではないかと思う．e-Japan戦略が実現した世界最高水準のネットワーク環境も遠からずメンテナンス・モードに入る．そのときに日本のメンテナンス技術が真価を発揮することを期待したい．

持続の条件：低エネルギ消費

原油価格がバレル$50近くまで上がってしまった．つい数年前まで$15程度だったわけだから，エネルギのコストが短期間で3倍以上に跳ね上がったことになる．以前から気になっていることなのだが，ネットワークの常時接続化が進んだためか，普通の生活でも情報化にかかわるエネルギ消費が急速に増えてきているように思える．ADSLや光ファイバを引くということは，モデムやDSUを常時オンにするということとほぼ同義である．そして話題のホーム・ネットワークである．このシステムは無線LANと各種端末が常時接続されることが暗黙の了解となっている．さらにネットワークの速度は映像伝送に耐える高速性が求められている．しかし，電子回路の常識では速度と消費電力は比例するのである．一瞬の映像伝送のために24時間一定の電力が消費され続けることになる．

そのため，ホーム・ネットワークでは，瞬間的な高速性とスタンバイ・モードの低消費電力性の二面性が重要になるはずである．究極は，スタンバイ時に消費電力が0であるようなネットワーク・アーキテクチャである．現在のインターネットのアーキテクチャはアプリケーションが動いていないときでも制御用のパケットが常時ネットワーク上を流れている構造であり，スタンバイというわけにもいかない．使わないときに電源を落とせばよいというものではない．多少のコストと常時接続のどちらを取るかと問われたら，常時接続が優先されるのが今の常識である．テクノロジは快適さを維持しつつ究極の性能を実現するためにある．

数年前に公開された映画にティム・バートン監督の「猿の惑星」がある．この映画では，数千年前に墜落した宇宙船を発掘し，そのパワー・スイッチを入れるとコンピュータが動きだすという設定がある．究極の持続可能技術は，文化や情報を数千年先に送ることもできるのである．

やまもと・つよし　北海道大学大学院情報科学研究科
メディアネットワーク専攻
情報メディア学講座

現実の事象を回路やソフトウェアで処理するための必須の技術

技術者のためのデータ計測

センサで捕らえたデータは，そのままA-D変換して使えることのほうが少なく，得られたデータを目的にあわせて抽出して計算し，目的に沿った形式に加工しなければならない．そして数値を計算する際には，有効数字の考え方や誤差の補正などが必要になってくる．もっとも重要な点は，「何を計測したいのか」をはっきりと認識し，正確に数値を収集することである．そのためには，測ろうとしている対象を十分に理解し，余計な数値を拾ってしまわないようにしなければならない．

今回の特集では，データの収集方法，そして得られたデータをいかに目的のために利用していくのかという点に焦点をあて，解説を行う．そして，実際に稼動しているデータ計測システムの考え方と概要を紹介する．

Prologue

何を測るのか，何を測りたいのか，どうやって測るのか

「測定する」ということ

宮崎 仁

● 何でも測れる時代

原子のサイズから銀河のサイズまで，近くのものから宇宙の果てまで，触れるものから触れないものまで，見えるものから見えないものまで，技術の進歩によってさまざまな量が測れるようになってきました．しかし，それだけに，うっかりすると本当に測りたい情報とは違う測定データを集めてしまう危険があることを忘れてはいけません．

たとえば象の大きさを測ってその外形を知りたいとき，1cm目盛りで10mぐらいの巻き尺を使って手作業で測定を行えば，目的に合った測定値が得られるでしょう．しかし，やろうと思えば体表のしわまでmm単位で測定したり，毛の1本までμm単位で測定することも，さらには細胞レベルや分子レベルまでnm単位で測定することだってできてしまいます（図1）．

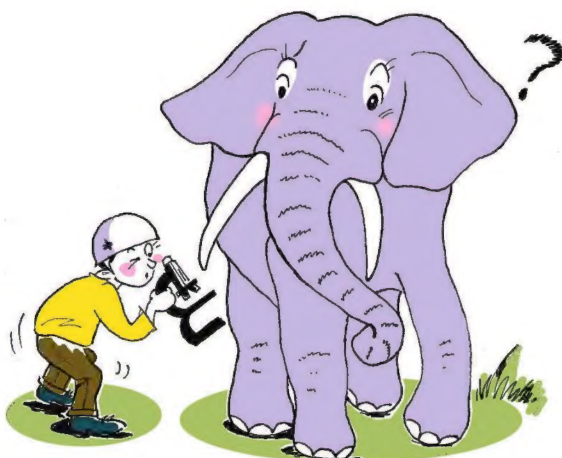


図1　確かに何でも測ることができるのだが……

人体の測定でも，ウェストが何cmか測ることも，しわや毛穴の大きさを測ることも，表皮細胞の大きさを測ることもできます．必要以上に細かい測定を行っても，データ量が膨大になるだけで，実用的なデータとはいえません．何を測りたいかを明確にして，目的に合わせて測定方法と測定レンジ（最小測定量～最大測定量）を決めることが大切です（図2）．

● 測定対象のゆらぎ

さらに，測定対象のゆらぎも十分考慮する必要があります．ウェストを測るときお腹を引き締めてしまった，などという人為的な大きなゆらぎは別としても，あらゆるアナログ量は多かれ少なかれゆらぎをもっています．散発的なゆらぎもありますし，周期的なゆらぎもあります．測定対象自体のゆらぎもありますし，電気的な方法で測定する場合には外部からのノイズの影響も強く受けます（図3）．

ゆらぎを考慮しないと，測るたびに測定値が違う（再現性がない）という問題を生じます．一方，ゆらぎの処理をうまく行えば，ゆらぎの中に埋もれた信号を取り出せる場合もあります．

ゆらぎの処理としては，ゆらぎを無視できる程度の粗い測定にとどめておく，積分回路などでアナログ的にゆらぎを平均化



図2　細かく測れば良いという話ではない

する，ゆらぎまできちんと測定してディジタル演算でゆらぎを平均化する，ゆらぎの性質（周期性など）に着目してゆらぎをキャンセルする，ゆらぎを含むありのままのデータを収集するなど，さまざまな方法があります．これまた，測定の目的に合わせて処理方法を選ぶことが大切です．

● 測定と A-D 変換

測定とは測りたい量をものさし（基準量）と比較して，もっとも近い値と対応付けることです．すなわち，測定値というのはつねに近似値です．真値に近づけることはできますが，真値そのものではありません．

また，測りたい量は無限桁のアナログ量であり，近似値は有限桁のディジタル量です．これは広い意味の A-D 変換（アナログ－ディジタル変換）といえます．ディジタル計測器では装置が自動的に近似（アナログ－ディジタル変換）を行いますが，アナログ計測器では人間が目盛りを読み取ることによって近似（アナログ－ディジタル変換）を行います．

測りたい量が直接比較しにくい量の場合には，前処理によって比較しやすい量に変換してから比較を行います．実際の測定では，最終的な比較・近似は電気量として行うのが普通です．

たとえば，測りたい量を電圧信号に変換し，基準量に対応する基準電圧と比較します．これは狭い意味の A-D 変換で，A-D コンバータで実行できます．そのための前処理では，測りたい量をセンサを用いて電気量に変換し，さらに電圧変換回路や増幅・補正回路，フィルタ回路などを用いて前処理を行います．

したがって，測定の誤差は大別して前処理の誤差，基準値（ものさし）の誤差，比較の誤差の三つに分けられ，基準値の誤差と比較の誤差は A-D 変換の誤差といえます．

● A-D 変換の誤差

A-D 変換の誤差は，また量子化誤差とそのほかのアナログ的誤差に分けられます．量子化誤差は分解能によって直接決まるもので，ものさしの最小目盛り，すなわちディジタル値の LSB に相当します．分解能が 8 ビットなら最小目盛り（1LSB）はフルスケール（FS）の 1/256 ≒ 0.39%FS の重みをもち，量子化誤差は±1/2LSB ≒±0.19%FS です．分解能が 16 ビットなら 1LSB は 1/65536 ≒ 0.0015%FS，量子化誤差は±1/2LSB ≒ 0.00076%FS となります．

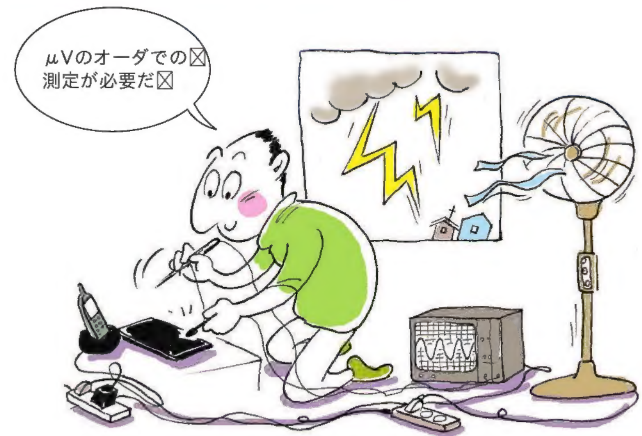


図3　ノイズの影響も考えなければならない

かりにものさしが理想的でアナログ的誤差がゼロだとしても，量子化誤差は必ず発生します．すなわち，量子化誤差はほかの誤差要因をゼロに近づけたときの誤差の極小値です．

量子化誤差は分解能を上げることでいくらでも小さくできます．しかし，実際の変換誤差は，量子化誤差とアナログ的誤差を合わせたものです．量子化誤差のほうが大きい場合は分解能を上げることで全体の誤差を小さくできます．しかし，アナログ的誤差のほうが支配的な場合は，それ以上分解能を上げても誤差の改善は頭打ちです．

また，アナログ的誤差が量子化誤差，すなわち±1/2LSB より小さければ，全体の誤差は± 1LSB の範囲に収まり，変換値はすべて有効といえます．しかし，アナログ的誤差が量子化誤差，すなわち±1/2LSB を上回る場合には，上位の桁にも誤差をもつ可能性があり，変換値はすべて有効とはいえません（図4）．

この場合のアナログ的誤差は，A-D 変換部分だけでなく，前



図4　誤差と有効数字

処理まで含めた全体のアナログ的誤差が影響してきます．A-D変換の分解能を1ビット増やすのは最近では難しいことではありませんが，それに対応してアナログ的誤差を1/2に低減するのは決して容易ではありません．

● A-D変換とサンプリング

A-D変換では1個の電圧値を基準値と比較して，ディジタル値に変換します．すなわち，1次元の変換を行います．測定対象が2次元以上の量，たとえば時間とか平面，空間などの次元をもつ量の場合，一般にその次元に沿ってサンプリング（スキャン）しながら連続的にA-D変換を行っていきます．このとき，サンプリングの粗さや誤差にも注意が必要です．

ディジタル・オーディオのような音声信号のA-D変換では，サンプリングによって電圧（振幅）方向と時間（周波数）方向の2次元の量をもつデータ列を収集します．このとき，時間方向の分解能はサンプリング周期（サンプリング周波数）で決まり，サンプリング周期よりも短時間のデータ変化は測定できません．また，前処理まで含めた時間方向のアナログ的誤差は，電圧方向の誤差と同様にオーディオ・データの誤差となります．

このサンプリング周期は，時間方向の分解能を決めるだけでなく，エイリアス（実際には存在しない低周波の偽情報）の発生原因となるので，十分な注意が必要です．サンプリング周波数の1/2以上の周波数成分が被測定信号に含まれる場合，その周波数成分は測定できないだけでなく，サンプリング周波数より低い周波数のエイリアスを生じます．

エイリアスを防ぐには，サンプリング周波数を測定したい信号周波数の2倍以上に選ぶとともに，被測定信号の不要な高周波成分はアンチエイリアス・フィルタで取り除きます．

エイリアスはオーディオ信号だけの現象ではなく，たとえばパソコン画面をビデオ・カメラで写したときに変な帯が現れたり，自動車などのタイヤを写したときに逆回転したり止まって見えるのもエイリアスの一種です．

● 波形歪みと高調波

電圧（振幅）と時間（周波数）の2次元量としてサンプリング・データを扱うときに，もう一つ注意が必要なのは，電圧方向の非直線性によって生じる高調波です．

測定系にジッタなど時間方向のアナログ的誤差があれば，測定データにも時間方向のアナログ的誤差を生じます．それによって，本来含まれていない周波数成分を生じます．

それだけでなく，測定系が電圧方向の非直線性誤差をもつ場合でも，測定データに波形歪みを生じます．歪んだ繰り返し波形には，もともとの波形にはない高調波成分が含まれており，振幅誤差が原因となって周波数誤差が生じたことになります．このように，サンプリングによる測定には，単なる1次元の測定とは異なる注意が必要です．

● 空間でのサンプリング測定

空間的な2次元形状，3次元形状を測定する場合にも，このようなサンプリング測定の注意は共通です．直線軸（X軸）に沿って一定間隔で直交方向（Y軸）の値をサンプリングすれば，2次元形状を測定できます．X軸，Y軸の2軸についてZ軸方向の値をサンプリングすれば，3次元形状を測定できます．

同様に一定間隔でスキャンしながら，寸法ではなくて紙面上の明るさ（色）を測定すれば，画像の取り込みになります．

これらの場合も，やはりエイリアスや高調波は発生します．たとえば，サンプリング間隔の1/2以上の周波数で測定形状や明るさが変化すれば，エイリアスを生じます．細かい模様の原稿をコピーすると，複製版に粗い変な模様が現れることがありますが，これも一般にエイリアスに相当します．空間方向のサンプリングの場合，時間方向と違って，被測定信号に簡単にフィルタをかけられないので，エイリアスはやっかいです．

コピー機のように元原稿と複製版を簡単に見比べられる場合はこのような問題が発生しても，すぐ気付くことができます．一般の測定の場合には，問題に気付かずそれが意味のあるデータだと思い込んでしまう危険が大きいので，より注意が必要でしょう．

● 本特集の構成

まず，第1章では，PCを用いてデータを測る実験を行ってみます．センサと入力ボードをつないでアプリケーション・プログラムを動かせば，いとも簡単にデータが取れるということを示します．

次の第2章では，「測定する」ということをいま一度考え直すため，「物理実験」にまで立ち戻り，データ計測の目的，データの測り方，誤差の考え方などを復習します．

そして第3章では，少ないデータ数や雑音を含むデータでも，なるべく精度よく種々の統計量や周波数スペクトル求める方法や雑音を抑圧する方法について，マルチレート信号処理やウェーブレット変換を用いて解説していきます．

たとえデータが正確に測定できても，後の解析に役立てなければ意味がなくなります．そこで第4章では，データを正確に取得することと同様に重要な，測定したデータを安全に貯め込むためのテクニックについて解説します．

第5章では，センサを使った「位置計測」の応用例として，「モーション・キャプチャ・システム」の実現例を示します．計測したデータを，どのように計算し，どのようにアプリケーションに生かすのか，その応用例として読んでください．

最後の第6章では，「何を測れば良いのか」を導き出す例として制御システムを取り上げます．制御しようとしているシステムのモデリングと数式化により，どんなデータが必要になるのか，しいてはどんなセンサが必要になるのか，また測れない値を計算で割り出すことすら可能だということが見えてくるはずです．

みやざき・ひとし　宮崎技術研究所

Chapter 1

とにかくデータ計測を試してみよう

PCとA-D変換ボードを使った温度計測実験

コンピュータで機器を制御するときに必要な基本技術は，A-D変換である．そこで，市販のボードでA-D変換を行い，センサからの情報を取り出してみることにより，簡単なデータ計測の実験を行ってみる．今回はそれをWindowsマシン上で行った．（筆者）

岸 哲夫

まず，今回の実験で使用した，ハードウェアの構成を示します．

A-D変換ボードを装着した，

マザーボード　：AOpen　AX3SPR
CPU　：Intel Pentium III 800EB
メモリ　：512Mバイト
ハードディスク：ATA100 6Gバイト
OS　：WindowsXP Pro SP1

というPCで実験を行いました．

A-D変換ボードのおもな仕様

今回はA-D変換ボードとして，ADM-686zPCI（マイクロサイエンス(株)，http://www.microscience.co.jp/）を使用しました．外観を写真1に示します．

この製品は，高速ブロックI/O転送命令も利用できるFIFOメモリを搭載しています．自動サンプリングを行うことが可能で，その際に指定されたクロック，トリガ，チャネル数に従って動作させることができます．



写真1　A-D変換ボードADM-686zPCIの外観

また，Windows 98/Me/2000/XP対応のハンドラ関数ライブラリとWDMデバイス・ドライバが付属しているため，このボードを使用したアプリケーションの作成が容易です．言語はそれぞれVisual C++，Visual Basic 5.0，Borland C 5.0，Delphi 3.0，C++ Builderに対応しています．

なお，Visual C++が手元になければ，無償で頒布されているBorland C++ Compiler 5.5を使用すると良いでしょう．ただし，開発環境はコマンド・ラインのみになります．

では，このADM-686zPCIの仕様を簡単に紹介します．

● アナログ入力部

- 入力数・信号形式
  16チャネル・シングル・エンド（普通の2線式），または8チャネル差動入力（スイッチ選択）
- 入力範囲
  ±10V/±5V/±2.5V/0～+10V/0～+5V（スイッチ選択）
- 入力インピーダンス
  各チャネルごとに10MΩの終端抵抗を標準実装．
- *CMRR*
  －65dB（差動入力のとき）
- クロストーク
  －82dB（各チャネル間）

● A-D変換部

- 分解能
  16ビット
- 単一チャネル・サンプリング速度
  5μs（200kHz）
- 複数チャネル・サンプリング速度
  5×（実行チャネル数）μs
- 非直線性
  ±0.004%FS

- 校正限度
  ±0.008 %FS
- 正確度1
  ±0.02　%FS
- 正確度2
  ±0.04　%FS
- 内部雑音
  ±4LSB
- 温度ドリフト
  ±10 ppm/℃
- A-D変換データ・コード
  バイナリ，または2の補数(ソフト指定)

● 制御部，その他

- クロック
  クロック源：内部10MHz/内部8.192MHz/外部TTL入力
  分周機能　：32ビット・プログラマブル・カウンタ(バイナリ)
- トリガ
  サンプリング開始
  内部トリガ：プログラム上からの即トリガ，アナログ入力(先頭チャネル)の指定エッジ，レベル，またはレンジ
  外部トリガ：外部TTL入力の指定エッジ，またはレベル



図1　IC温度センサLM35Dの仕様

- バッファ・メモリ
  標準1024ワードFIFOメモリ
- データ転送
  ブロック転送：通常，FIFOのHALF-FULLフラグを利用して容量の半分単位で行う
  通常IN命令：2バイト(上位・下位)に分割して連続読み込み
- マスタ-スレーブ動作
  マスタのクロック出力をスレーブのクロック源入力に接続することにより可能
- 割り込み
  割り込み要因：1回サンプリング・スキャン終了，トリガ発生，サンプリング・クロック，外部TTL入力の指定エッジ，FIFOメモリのEMPTY解消，同HALF-FULLフラグ
- 汎用ディジタル入出力
  1ビット・TTL入力，1ビット・5Vロジック出力(またはオープン・コレクタ)
- I/Oアドレス
  組み込み対象システムのプラグ&プレイ機能により(連続した)16アドレス占有

試験用センサと信号増幅用アンプ

このA-D変換ボードは，アナログ入力とディジタル入力の両方に対応していますが，今回はアナログ・タイプのセンサを使用することにします．つまり事象が発生すると，電圧の変化が起き，それをA-D変換ボードを通じてPCに記録します．

数多くのセンサが市販されていますが，今回は安価な「温度センサ」であるLM35Dを使うことにします．

LM35Dの仕様は，http://www.national.com/JPN/ds/LM/LM35.pdfで入手することができます(図1)．

この製品はナショナルセミコンダクター社の製品で，扱いやすく需要が多い製品です．

LM35シリーズは出力電圧が温度に直線的に比例する特性をもつ高精度IC温度センサです．したがって温度を電圧に変換することが容易です．LM35は単一電源または±両電源が使用可能です．電源からは60μAの電流が流れるだけなので，自己発熱は少なく，静止空気で0.1℃以下の温度上昇です．LM35は−55℃から+150℃の温度範囲で動作します．

蛇足ですが，おもしろい温度センサの使用法が記してあるWebサイトを見つけました(図2)．納豆の発酵温度の管理にセンサを使っているようです．

こういう用途の工作ができると，人生の幅が広がります．

● センサの電圧変化をOPアンプで増幅

このセンサをA-D変換ボードの入力とするには，かりに入力範囲を±2.5Vに設定したとしても，1℃の変化で10mVの電圧

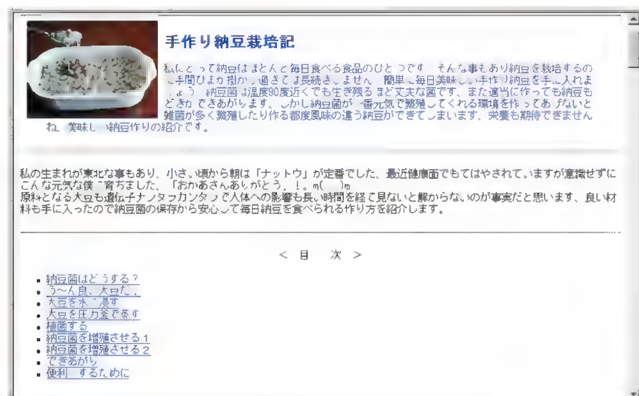


図2　手作り納豆栽培記（http://www.geocities.co.jp/NatureLand/9485/nat.html）

変化では，検出電圧が低すぎて問題があります．そこでOPアンプで増幅することにします．

このような用途では，一般的な非反転増幅回路が使いやすいのでこの回路を使用します．今回は5倍に増幅しました．

LM358NはOPアンプ回路が2回路入っていて，単電源で動作するので，このような用途に向いていると思います．電源電圧範囲は単電源で+3～30Vです．

● マイクロサイエンスの4チャネル個別ゲイン・アンプ

今回の実験では，たくさんの温度センサを接続する必要はないので，A-D変換ボード「ADM686zPCI」のオプションである，4チャネル個別ゲイン・アンプ「BGA-305BRD」（写真2）を使用します．

この製品はスイッチで1倍，5倍，任意倍の増幅度の設定ができます．今回は5倍で使用します．

このボードはA-D変換ボード「ADM686zPCI」に取り付けて使用します．抜き差しの際にピンが曲がることのないように注意してセットしましょう．

ゲイン切り替えスイッチは1チャネルごとに4個あります．これらはすべて5倍の場所にセットします．

A-D変換ボードを使用する準備

37ピン端子台接続アダプタ「CTML-37」を使用して，センサ接続側の配線を行うことにします．直接センサとつなぐと不便なので，あいだに弱電用のターミナル（写真3）を付けることにします．

念のために書き加えておきますが，端子につなぐリード線の先ははんだで処理しておきます．筆者は最近ガスボンベ式はんだごてを使っていて，細かい作業の際にはこれが便利だと思います．

● LM35Dの接続方法

温度センサLM35Dのピン配置を図3に，端子台接続アダプタCTML-37のピン配置を図4に示します．

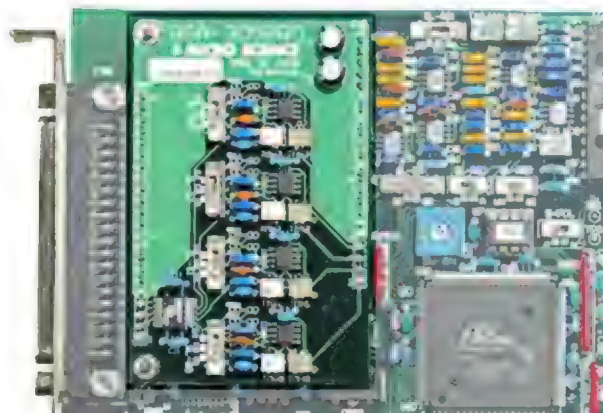


写真2　4チャネル個別ゲイン・アンプBGA-305BRD

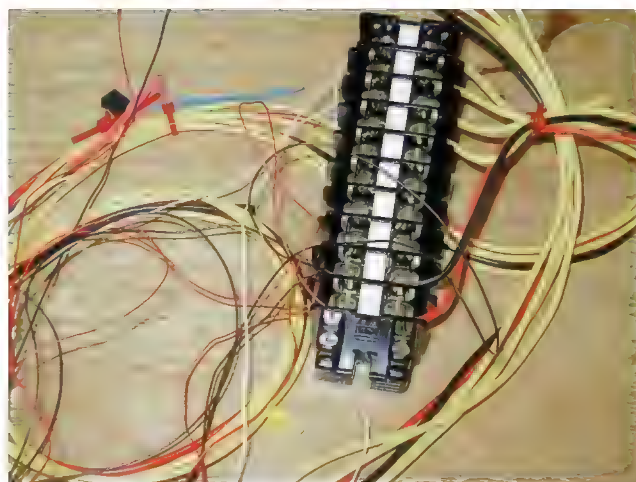


写真3　弱電用のターミナル

$+V_s$は電源（+5V）に接続します．またGNDはアースに接続し，V_{OUT}はチャネル0から3に接続します．

● A-D変換ボードをPCに取り付ける

取り付けは簡単です．PCIスロットに差し込むだけです．再起動すれば認識されるはずです．

取り付け直後のデバイス・マネージャのようすを図5に，ドライバのインストールのようすを図6に，ドライバをインストールした後のデバイス・マネージャのようすを図7に示します．

その後，電源を落とし，37ピン端子台接続アダプタをボードのピンに接続して，再起動してください．

● 動作確認を行う

まず，A-D変換ボードに同梱されたCD中の「動作確認」フォルダにある`td686w2.exe`を実行します．コマンド・プロンプト上で起動させます．

すべてのチャネルにセンサをすべてつないでいなければ，マニュアル・サンプリング後のステータスやデータ読み取り後のステータスでエラーになりますが，その他のエラーがなければ正常です（図8）．

New Products――TI，5V単一電源動作のOPアンプ「THS4304」を発売～低雑音，低歪みで高いダイナミック・レンジ
米Texas Instruments社は，5V単一電源動作で，静的消費電流が18mAのOPアンプ「THS4304」を発売した．ユニティ・ゲイン帯域は3GHz，スルー・レートは830V/μs，入力雑音は2.4nV/√Hz．電圧帰還方式を採用している．1,000個購入時の単価は￥250.

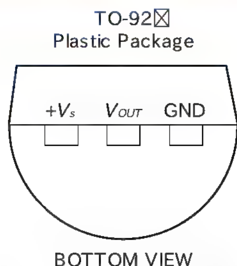


図3 LM35D のピン配置図

図4 CTML-37 のピン配置図

信号名	機 能	ピン番号	ピン番号	信号名（機能）
CH0（0H）	CH0入力（差動：ch0の＋側）	1	20	AG（アナログ・グラウンド）
CH1（0L）	CH1入力（差動：ch0の－側）	2	21	AG（アナログ・グラウンド）
CH2（1H）	CH2入力（差動：ch1の＋側）	3	22	AG（アナログ・グラウンド）
CH3（1L）	CH3入力（差動：ch1の－側）	4	23	AG（アナログ・グラウンド）
CH4（2H）	CH4入力（差動：ch2の＋側）	5	24	AG（アナログ・グラウンド）
CH5（2L）	CH5入力（差動：ch2の－側）	6	25	AG（アナログ・グラウンド）
CH6（3H）	CH6入力（差動：ch3の＋側）	7	26	AG（アナログ・グラウンド）
CH7（3L）	CH7入力（差動：ch3の－側）	8	27	AG（アナログ・グラウンド）
CH8（4H）	CH8入力（差動：ch4の＋側）	9	28	AG（アナログ・グラウンド）
CH9（4L）	CH9入力（差動：ch4の－側）	10	29	AG（アナログ・グラウンド）
CH10（5H）	CH10入力（差動：ch5の＋側）	11	30	AG（アナログ・グラウンド）
CH11（5L）	CH11入力（差動：ch5の－側）	12	31	AG（アナログ・グラウンド）
CH12（6H）	CH12入力（差動：ch6の＋側）	13	32	AG（アナログ・グラウンド）
CH13（6L）	CH13入力（差動：ch6の－側）	14	33	AG（アナログ・グラウンド）
CH14（7H）	CH14入力（差動：ch7の＋側）	15	34	AG（アナログ・グラウンド）
CH15（7L）	CH15入力（差動：ch7の－側）	16	35	AG（アナログ・グラウンド）
	空ピン	17	36	空ピン
S/H	S/H信号出力	18	37	DG（ディジタル・グラウンド）
＋5V	PCIバス上の＋5V電源出力	19		

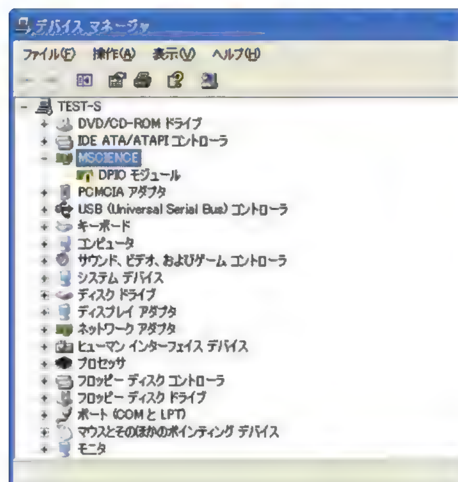
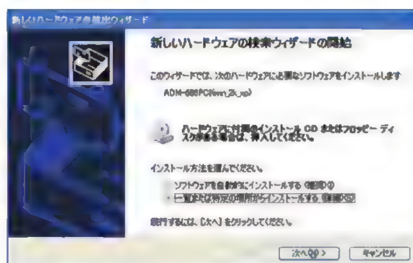
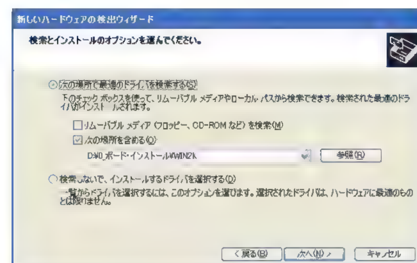


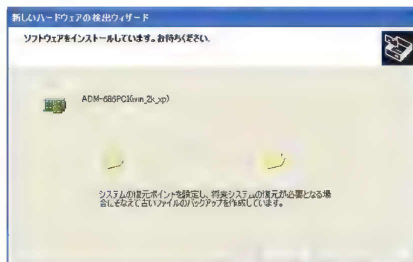
図5 ボード取り付け直後のデバイス・マネージャ



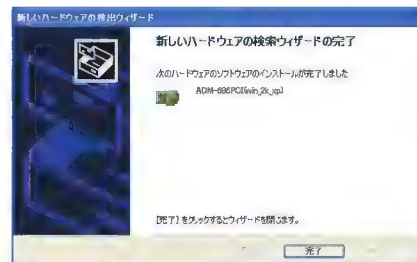
(a)



(b)



(c)



(d)

図6 ドライバのインストール

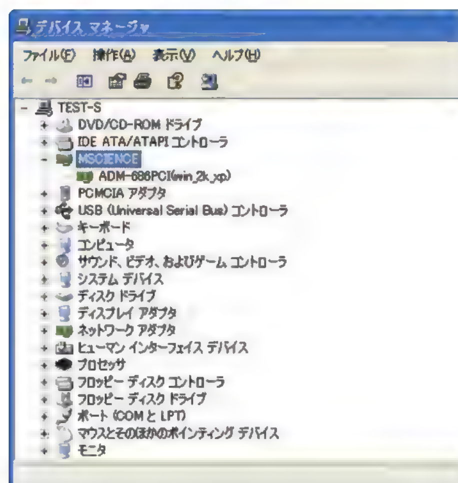


図7 ドライバをインストールした後のデバイス・マネージャ

図8 td686w2.exe の実行結果

```
ADM-686PCI を１枚検出しました。
１枚目のボードのチェックをします。
ボード ID(0fH)は正常で、ボード番号設定は(0)です。
リセット時のステータス(02H)は正常です。
マニュアル・サンプリング後のステータス(02H)は異常です。
データ読み取り後のステータス(8aH)は異常です。
１０MHz クロックのテスト中... 終了しました。
クロック数(10)、データ数(10)は正常です。
８．１９２MHz クロックのテスト中... 終了しました。
クロック数(10)、データ数(10)は正常です。
サンプリング中... 終了しました。
サンプリング後のステータス(adH)は正常です。
フラグ・クリア後のステータス(0dH)は異常です。
ＦＩＦＯカウント中... 終了しました。
FULL =( 1029) HALF = (512)です。
フラグは(8eH)です。
```

New Products――アドイン研究所，VoIP 多点間接続サーバ・ソフトウェア「IntelliVoice Mint Square」を発売
（株）アドイン研究所は，多点間接続機能を備えた SIP プロトコル・ベースの VoIP サーバ・ソフトウェア「IntelliVoice Mint Square」を発売した．http://www.adin.co.jp/

図 9　ts686w2.exe の実行結果

```
ADM-686PCI を 1 枚検出しました。
8d19 8d7f 8ce0 7867 7d58 7ef1 7f69 7f95 7fd6 7fdd 7fcf
7fcc 7fd9 7fcf 7fdd 7fd8 8d37 8d86 8cf3 789f 7cd3 7ee3
7fb8 8008 806f 807e 806e 8077 8084 807f 8089 8085 8cfa
8d94 8cf2 792c 7cbe 7ed6 7fd9 8047 80c8 80e4 80d4 80e2
80ee 80ec 80fb 80f3 8d4f 8d7a 8cfe 7997 7cd9 7ed9 7feb
8069 80fd 811e 8112 8125 8134 8134 8141 813c 8d08 8d8f
8cde 79f2 7d03 7ee8 7ff9 807a 811a 813f 8138 814d 815e
815e 816f 8168 8d1e 8d93 8d03 7a6c 7d40 7f02 8007 8089
812d 8155 8153 8162 8178 817c 8189 8185 8d0e 8d7b 8cf0
7aed 7d7e 7f22 8017 8097 8137 8162 8164 8172 8188 818b
819a 8198 8d22 8d95 8cf0 7b7b 7dc8 7f48 802c 80a3 8140
816a 816b 817b 8192 8197 81a7 81a4 8d17 8d80 8d08 7bea
7e0e 7f72 8042 80b2 8148 8170 8170 8183 819a 819d 81b1
81ab 8d21 8d86 8cea 7ca9 7e6f 7fa0 805e 80bd 814f 8177
8176 8188 819e 81a0 81b5 81b1 8d37 8d80 8cf0 7d4a 7ed0
7fd6 807c 80d0 815c 817c 8178 818c 81a1 81a4 81b7 81b8
8cf7 8d8f 8cf3 7e18 7f3e 8013 809d 80e4 8166 8185 817f
818d 81a4 81a8 81ba 81b7 8d5c 8d7d 8ce1 7ee8 7fb3 8052
80c2 80f8 8171 8189 8184 8190 81a8 81a8 81bd 81b8 8d1a
8d78 8cf4 7fe6 8033 809c 80e9 8111 817b 818f 8186 8194
81aa 81a9 81bd 81ba 8cf9 8d9a 8cfd 80ed 80c9 80ec 8119
8127 818b 819a 818b 8198 81ac 81aa 81bf 81ba 8d2e 8d6f
8d06 81c0 8151 813d 8146 8141 819b 81a4 8191 819b 81ad
81aa 81c1 81bd 8cf2 8d99 8ce9 82d1 81e6 8191 8179 815e
81ab 81aa 8196 819e 81b0 81ab 81c1 81be 8d36 8d7e 8d00
83a3 826e 81e6 81a6 817b 81ba 81b4 819c 819f 81b2 81b1
81c1 81be 8d22 8d7d 8cfe 848f 82f0 8235 81d7 8197 81cc
81be 81a1 81a4 81b4 81b1 81c3 81bf 8d2c 8d87 8cd8 852a
8364 827d 8205 81b4 81db 81cc 81a8 81aa 81b6 81af 81c2
81bf 8d38 8d72 8d05 85d6 83d0 82bf 822f 81cd 81ec 81d3
81ac 81ab 81b4 81b3 81c4 81c0 8d34 8d8e 8ce8 8678 8432
82fe 8258 81e6 81fa 81db 81b2 81ad 81b7 81b4 81c5 81c0
8d4b 8d71 8ce1 86c1 8477 832f 827a 81fa 8209 81e5 81b9
81b2 81bb 81b6 81c4 81c1 8cf3 8d9e 8cf7 873e 84bb 835d
8295 820d 8215 81ed 81bb 81b4 81be 81b5 81c4 81c2 8d43
8d7a 8cfb 876c 84ea 837f 82af 8220 8221 81f4 81c0 81b6
81c0 81b5 81c6 81c3 8d06 8d9c 8cfc 87de 8524 83a5 82c5
822f 822a 81fa 81c4 81ba 81bf 81b5 81c5 81bf 8d2f 8d8e
8ce6 883c 8564 83c6 82dc 823c 8232 8202 81c9 81bb 81c1
81bb 81c8 81c2 8d34 8d76 8cf1 886e 858b 83e4 82ef 8245
823c 8205 81cc 81c0 81c3 81b8 81c8 81c3
```

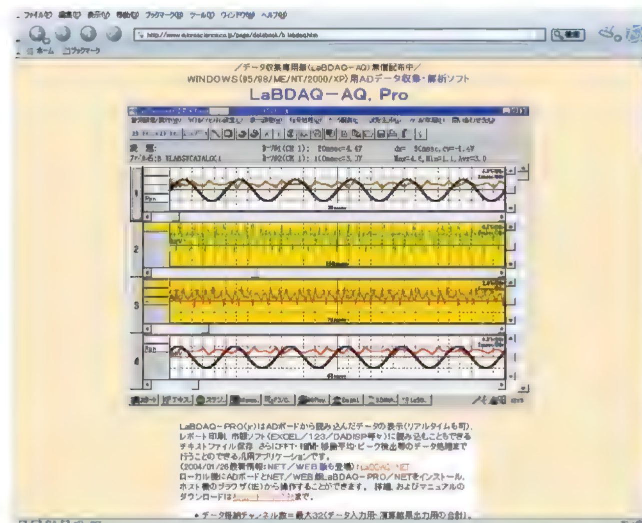


図 10　LaBDAQ-AQ のダウンロード・ページ

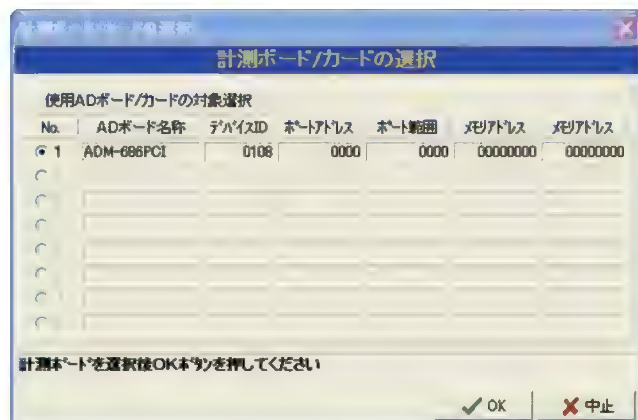


図 11　計測ボード /カード の選択

次に同じ場所にある ts686w2.exe を実行してみます．センサからデータを読み込み，16 進数で表示しています（図 9）．何かデータが読み込めれば正しく動作しているはずです．

実際にデータを読み込む

プログラム中で使用できるライブラリと関数が A-D 変換ボードに同梱された CD に入っています．しかし，それだけでアプリケーションを作成するには，少し手間がかかります．

ここでは無償のデータ収集・解析アプリケーションである LaBDAQ-AQ をダウンロードして，使用してみましょう．

http://www.microscience.co.jp/page/databook/b_labdaq.htm（図 10）

からダウンロードします．

インストールは setup を実行するだけです．起動したら「計測ボード /カード の選択」を行います（図 11）．このダイアログで「ADM－686PCI」が選択できなかったら，ボードが認識されていないので，再度確認してください．

全体の画面は図 12 のようになります．

まず，計測実行条件の設定を行います（図 13）．ここではサンプリング速度と分周比を指定しています．0.5s 間隔で 100 回データを取ります．

次にチャネル設定を行います（図 14）．サンプリング・チャネル数を 4 に設定し，サンプリング・データ点数を 100 に設定しています．

その後スタートさせるとデータを取得します．

取得したデータは「全チャンネルデータ編集」で見ることができます（図 15）．

取得したデータは，図 16 に示すとおりです．

このデータを保存するには「現在データのテキストファイル保存」を実行します（図 17）．CSV やタブ区切りで保存することができ，後で Excel などで加工することが可能です．

今回はセンサ 1（チャネル 0）に保冷剤を密着させ，センサ 2（チャネル 1）はお湯の入った器の底に密着させ，センサ 3（チャ

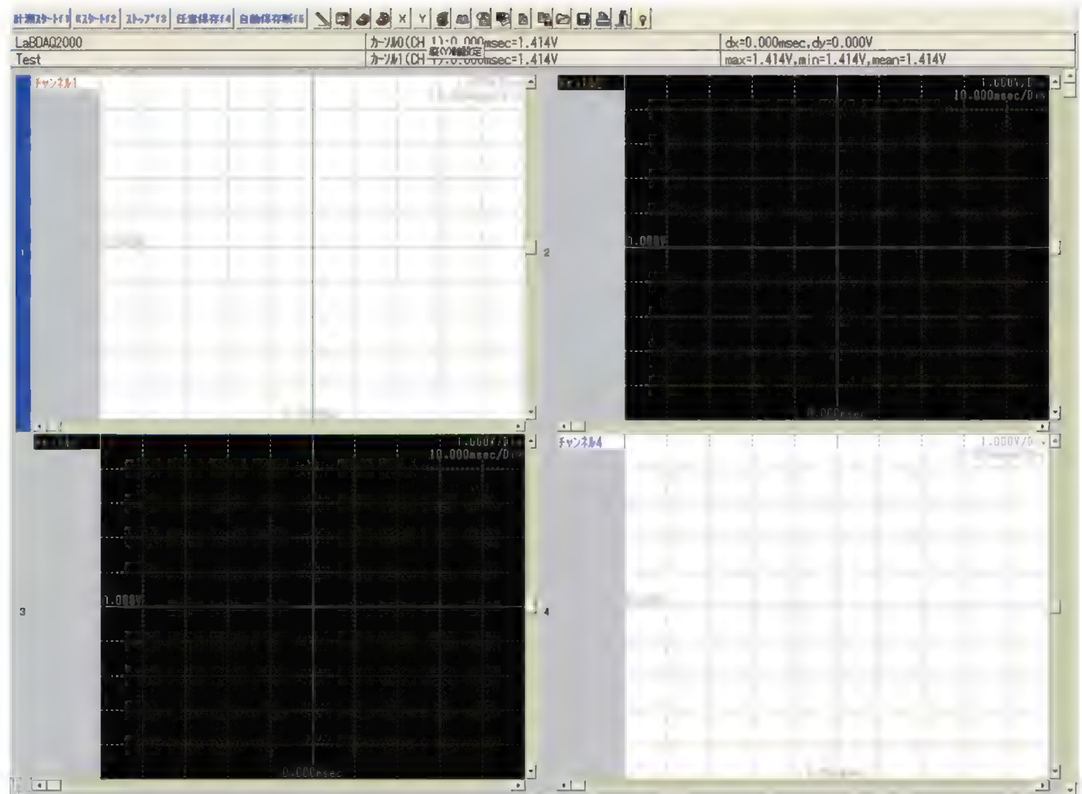


図12
LaBDAQ-AQ全体画面

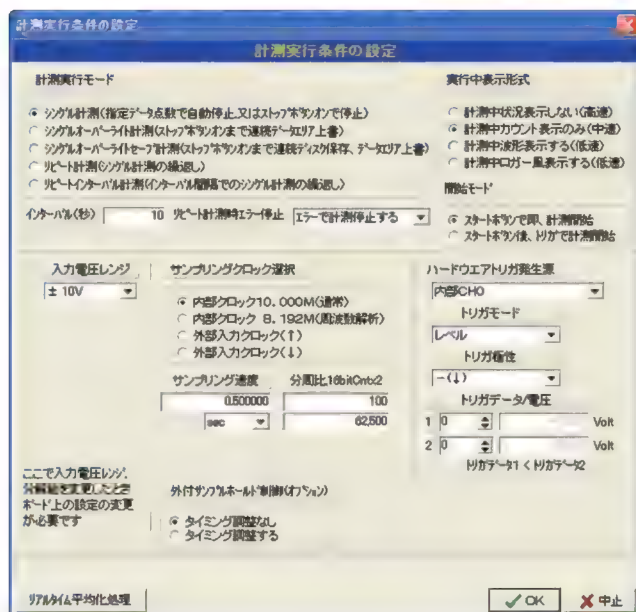


図13　計測実行条件の設定



図14　チャンネル設定の画面

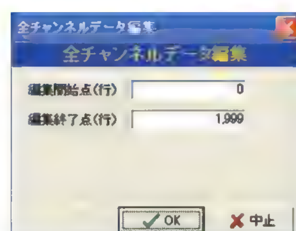


図15
全チャンネルデータ編集

全チャンネルデータ編集

No	ﾁｬﾝﾈﾙ1	ﾁｬﾝﾈﾙ2	ﾁｬﾝﾈﾙ3	ﾁｬﾝﾈﾙ4	ﾁｬﾝﾈﾙ5	ﾁｬﾝﾈﾙ6	ﾁｬﾝﾈﾙ7	ﾁｬﾝﾈﾙ8	ﾁｬﾝﾈﾙ9
0	0.515	2.218	1.300	0.230	0.000	0.000	0.000	0.000	0.00
1	0.290	2.202	1.286	0.000	0.000	0.000	0.000	0.000	0.00
2	0.270	2.211	1.293	0.000	0.000	0.000	0.000	0.000	0.00
3	0.909	2.208	1.315	0.670	0.000	0.000	0.000	0.000	0.00
4	0.971	2.226	1.312	0.000	0.000	0.000	0.000	0.000	0.00
5	0.554	2.233	1.276	0.000	0.000	0.000	0.000	0.000	0.00
6	0.225	2.214	1.292	0.262	0.000	0.000	0.000	0.000	0.00
7	0.579	2.232	1.280	0.538	0.000	0.000	0.000	0.000	0.00
8	0.905	2.225	1.328	0.000	0.000	0.000	0.000	0.000	0.00
9	0.313	2.240	1.311	0.000	0.000	0.000	0.000	0.000	0.00
10	0.564	2.235	1.322	0.485	0.000	0.000	0.000	0.000	0.00
11	0.178	2.248	1.297	0.410	0.000	0.000	0.000	0.000	0.00
12	0.246	2.193	1.298	0.000	0.000	0.000	0.000	0.000	0.00
13	0.265	2.202	1.286	0.000	0.000	0.000	0.000	0.000	0.00
14	0.558	2.234	1.290	0.598	0.000	0.000	0.000	0.000	0.00
15	0.332	2.268	1.297	0.180	0.000	0.000	0.000	0.000	0.00
16	0.332	2.250	1.295	0.000	0.000	0.000	0.000	0.000	0.00
17	0.199	2.251	1.309	0.000	0.000	0.000	0.000	0.000	0.00
18	0.159	2.258	1.306	0.694	0.000	0.000	0.000	0.000	0.00
19	0.409	2.238	1.316	0.000	0.000	0.000	0.000	0.000	0.00
20	0.576	2.249	1.285	0.000	0.000	0.000	0.000	0.000	0.00
21	0.676	2.237	1.314	0.156	0.000	0.000	0.000	0.000	0.00
22	0.905	2.236	1.301	0.597	0.000	0.000	0.000	0.000	0.00

行挿入　行削除　設定　表印刷　更新後終了　中止

図16　取得したデータ

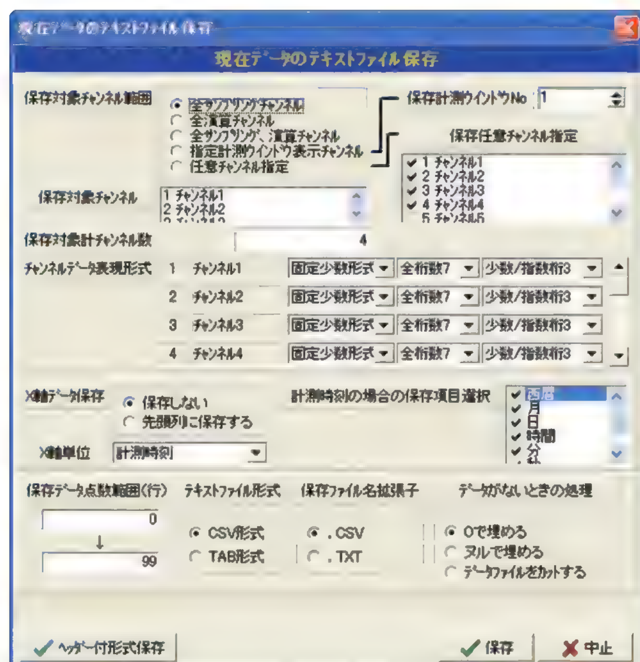


図 17　現在データのテキストファイル保存画面

ネル 2)は何もせず，チャネル 3にはセンサを接続しないで実行しました．その実行結果を図 18に示します．

まず，センサ1(チャネル 0)ですが，センサが敏感なため保冷剤と机の間の温度を計測しているようです．値が上がり下がりしています．

次にセンサ2(チャネル 1)ですが，お湯の入った器の底の温度が徐々に上がっています．44～47℃近くまで上昇しています．

次にセンサ3(チャネル 2)は室温のままです．26℃近辺を指しています．

そして，何も接続していない，チャネル 3の値はノイズとなっています．

では次に計測中に大きな温度変化をさせてみます．

室温，お湯から冷却，室温から冷却の 3種類を試してみました．冷却には「エアダスター」を使いました．結果を図 19に示します，問題なく，温度変化を認識しています．もっとも 0.5秒間隔で取得しているデータなので当然ですが．

では，次に 0.5ms 間隔でサンプリングして，大きな温度変化をさせてみます(図 20)．

この A-D 変換ボードを使用すれば，Windows でも，ある程度のリアルタイム性能を確保できることがわかりました．

問題は OS がハングアップしたときに，センサが役に立たなくなることです．安定したマシンを使用すれば，ホーム・セキュリティにも応用可能だということがわかりました．

図 18　実行結果

電圧				温度に変換		
チャネル 0	チャネル 1	チャネル 2	チャネル 3	保冷剤	お湯	室温
0.515	2.218	1.3	0.23	10.3	44.36	26
0.29	2.202	1.296	− 0.604	5.8	44.04	25.92
0.27	2.211	1.293	− 0.082	5.4	44.22	25.86
0.909	2.208	1.315	0.67	18.18	44.16	26.3
0.971	2.226	1.312	− 0.068	19.42	44.52	26.24
0.554	2.233	1.276	− 0.634	11.08	44.66	25.52
0.225	2.214	1.292	0.262	4.5	44.28	25.84
0.579	2.232	1.28	0.538	11.58	44.64	25.6
0.805	2.225	1.328	− 0.38	16.1	44.5	26.56
0.313	2.24	1.311	− 0.492	6.26	44.8	26.22
0.564	2.235	1.322	0.485	11.28	44.7	26.44
0.178	2.248	1.297	0.41	3.56	44.96	25.94
0.246	2.193	1.298	− 0.491	4.92	43.86	25.96
0.265	2.202	1.286	− 0.273	5.3	44.04	25.72
0.558	2.234	1.29	0.598	11.16	44.68	25.8
0.332	2.268	1.297	0.18	6.64	45.36	25.94
0.332	2.25	1.295	− 0.647	6.64	45	25.9
0.199	2.251	1.309	− 0.075	3.98	45.02	26.18
0.159	2.258	1.306	0.694	3.18	45.16	26.12
0.408	2.238	1.316	− 0.086	8.16	44.76	26.32
0.576	2.249	1.285	− 0.628	11.52	44.98	25.7
0.676	2.237	1.314	0.156	13.52	44.74	26.28
0.905	2.236	1.301	0.597	18.1	44.72	26.02
0.776	2.247	1.299	− 0.304	15.52	44.94	25.98
0.244	2.255	1.3	− 0.504	4.88	45.1	26
0.342	2.274	1.308	0.349	6.84	45.48	26.16
0.548	2.247	1.271	0.476	10.96	44.94	25.42
0.374	2.278	1.298	− 0.461	7.48	45.56	25.96
0.214	2.265	1.309	− 0.414	4.28	45.3	26.18
0.26	2.223	1.291	0.459	5.2	44.46	25.82
0.173	2.258	1.284	0.456	3.46	45.16	25.68
0.216	2.268	1.298	− 0.482	4.32	45.36	25.96
0.435	2.254	1.325	− 0.424	8.7	45.08	26.5
0.353	2.252	1.312	0.504	7.06	45.04	26.24

図 19　温度変化させた計測結果

電圧				温度に変換		
チャネル 0	チャネル 1	チャネル 2	チャネル 3	室温	お湯→冷却	室温→冷却
1.357	1.675	1.371	-0.806	27.14	33.5	27.42
1.351	1.72	1.372	-0.108	27.02	34.4	27.44
1.376	1.749	1.373	0.848	27.52	34.98	27.46
1.345	1.8	1.373	-0.244	26.9	36	27.46
1.375	1.826	1.366	-0.717	27.5	36.52	27.32
1.368	1.854	1.384	0.392	27.36	37.08	27.68
1.366	1.887	1.376	0.725	27.32	37.74	27.52
1.346	1.903	1.381	-0.396	26.92	38.06	27.62
1.361	1.928	1.375	-0.751	27.22	38.56	27.5
1.377	1.95	1.382	0.194	27.54	39	27.64
1.35	1.976	0.038	0.981	27	39.52	0.76
1.365	1.996	-0.025	0.05	27.3	39.92	-0.5
1.335	2.014	-0.006	-1.034	26.7	40.28	-0.12
1.374	2.019	0.01	0.133	27.48	40.38	0.2
1.378	1.983	0.012	0.958	27.56	39.66	0.24
1.369	1.938	0.001	0.186	27.38	38.76	0.02
1.357	1.427	-0.058	-0.779	27.14	28.54	-1.16
1.325	-0.009	-0.045	-0.96	26.5	-0.18	-0.9
1.39	0.001	0.142	0.852	27.8	0.02	2.84
1.363	0	0.036	0.783	27.26	0	0.72
1.356	-0.035	-0.001	0.076	27.12	-0.7	-0.02
1.378	-0.031	0.023	-0.724	27.56	-0.62	0.46
1.37	-0.009	0.018	-0.558	27.4	-0.18	0.36
1.366	-0.003	0.083	0.369	27.32	-0.06	1.66
1.373	-0.018	0.036	0.882	27.46	-0.36	0.72
1.378	-0.008	0.061	0.441	27.56	-0.16	1.22
1.344	-0.018	-0.028	-0.641	26.88	-0.36	-0.56
1.377	-0.01	0.122	-0.8	27.54	-0.2	2.44
1.37	0.045	0.201	0.154	27.4	0.9	4.02
1.335	0.043	0.46	0.857	26.7	0.86	9.2
1.387	0.016	0.658	0.052	27.74	0.32	13.16

図 20　サンプリング間隔を狭くした結果

チャネル 0	チャネル 1	チャネル 2	チャネル 3	室温	お湯→冷却	室温
1.361	2.538	1.385	0.051	27.22	50.76	27.7
1.379	2.542	1.372	0.082	27.58	50.84	27.44
1.366	2.536	1.372	0.39	27.32	50.72	27.44
1.347	2.536	1.383	-0.882	26.94	50.72	27.66
1.356	2.536	1.376	-0.394	27.12	50.72	27.52
1.377	2.535	1.372	0.915	27.54	50.7	27.44
1.383	2.529	1.373	0.388	27.66	50.58	27.46
1.369	2.53	1.381	-0.883	27.38	50.6	27.62
1.365	2.532	1.374	-0.395	27.3	50.64	27.48
1.353	2.53	1.388	0.921	27.06	50.6	27.76
1.334	2.54	1.376	0.402	26.68	50.8	27.52
1.358	2.538	1.371	-0.881	27.16	50.76	27.42
1.352	2.538	1.372	-0.402	27.04	50.76	27.44
1.37	2.534	1.372	0.9	27.4	50.68	27.44
1.363	2.533	1.38	0.378	27.26	50.66	27.6
1.356	2.529	1.392	-0.88	27.12	50.58	27.84
1.359	2.526	1.387	-0.397	27.18	50.52	27.74
1.355	2.529	1.374	0.905	27.1	50.58	27.48
1.355	2.533	1.37	0.392	27.1	50.66	27.4
1.359	2.538	1.371	-0.864	27.18	50.76	27.42
1.365	2.533	1.374	-0.384	27.3	50.66	27.48
1.356	2.531	1.381	0.9	27.12	50.62	27.62
1.365	2.531	1.38	0.381	27.3	50.62	27.6
1.356	2.528	1.386	-0.861	27.12	50.56	27.72
1.346	2.531	1.371	-0.381	26.92	50.62	27.42
1.35	2.536	1.374	0.885	27	50.72	27.48
1.372	2.535	1.375	0.38	27.44	50.7	27.5
1.366	2.528	1.387	-0.862	27.32	50.56	27.74
1.345	2.53	1.378	-0.399	26.9	50.6	27.56
1.328	2.535	1.378	0.886	26.56	50.7	27.56
1.381	2.534	1.377	0.382	27.62	50.68	27.54
1.367	2.53	1.374	-0.86	27.34	50.6	27.48



付属のライブラリ関数について

付属のライブラリ関数の使い方を検証してみます．今回は無償で使用できる Borland C を使用するので，話をややこしくしないため GUI は一切使いません．

したがって，GUI 操作を含むライブラリ関数は，説明だけにとどめます．

関数名の一覧を表 1 に示します．

このライブラリ関数をラッピングした関数ライブラリが PORT_2K.LIB です．

この関数群は ADM-686zPCI が一枚，または複数枚インストールされている環境で有効です．

表 1　関数名一覧

AD_Open_ADSys	ボード，および本ハンドラの初期化
AD_Set_SampCh	サンプリング実行チャネル関連設定実行チャネル数
AD_Set_SampMode	サンプリング・モード，イベントの設定 A-D データ転送先，方法
AD_Set_Trigger	トリガ関連設定（レンジ・トリガ以外）トリガ源，レベル，モード
AD_Set_RangeTrigger	トリガ関連設定（レンジ・トリガ）
AD_Set_MultiSamp	マルチサンプリング動作モード設定
AD_Set_Exclk	オプション，外部クロック源の設定クロック源の周波数値
AD_Set_Mexclk	同上マルチサンプリング・クロック源用クロック源の周波数値
AD_Set_Clock	サンプリング・クロックの設定クロック源，周期値，単位
AD_Set_MsClock	同上マルチサンプリング・クロック用クロック源，周期値，単位
AD_Start_Samp	サンプリング開始（トリガ待ち or 即）サンプリング点数
AD_Get_Status	ステータス取得サンプリング進捗状況など
AD_Read_DllData	DLL 内バッファからデータ読み出し A-D データ格納バッファ
AD_Read_DirectFifo	A-D ボードから直接にデータ読み出し A-D データ格納バッファ
AD_Get_OneScan	マニュアル（1回）サンプリング
AD_Close_ADSys	本ハンドラの終了
AD_Out_Aux	汎用ディジタル（ラッチ）出力データ
AD_Inp_Aux	汎用ディジタル（現在値）入力データ
AD_Set_SampLoop	データ・バッファをリング状に設定．一周後は上書き
AD_Set_Inpmode	データ・コード指定
AD_Stop_Samp	サンプリング動作の（強制）中止
AD_Read_RestData	A-D ボードの残りデータ読み込みエラー停止の後
AD_Clear_Flags	A-D ボードのフラグ・クリア・ビット指定
AD_Set_ScanSpeed	サンプリング・スキャン速度指定
AD_Set_Shc	外付けサンプル&ホールド制御オプション
AD_Plus_Message	オプション的なメッセージ発信の設定任意
AD_Get_Libver	バージョン情報取得

● ラッピングされている関数名一覧

```
Open_Driver( int device_id[] );
Close_Driver( void );
Start_Driver( int board_no, int base_no,
                          int *resouce_no );
Stop_Driver( int resouce_no );
ReadByte( int resouce_no, int address );
ReadWord( int resouce_no, int address );
ReadDWord( int resouce_no, int address );
ReadByteBlock( int resouce_no, int address,
                   int count, UCHAR *pbuf );
ReadWordBlock( int resouce_no, int address,
                  int count, USHORT *pbuf );
ReadDWordBlock( int resouce_no,
     int address, int count, ULONG *pbuf );
WriteByte( int resouce_no, int address,
                              UCHAR data );
WriteWord( int resouce_no, int address,
                             USHORT data );
WriteDWord( int resouce_no, int address,
                              ULONG data );
WriteByteBlock( int resouce_no,
     int address, int count, UCHAR *pbuf );
WriteWordBlock( int resouce_no,
    int address, int count, USHORT *pbuf );
WriteDWordBlock( int resouce_no,
     int address, int count, ULONG *pbuf );
GetMemoryAddress( int resouce_no,
                       int *memaddress );
```

● Open_Driver(int device_id[]);

この関数はint 8個分の配列に，初期値として-1をセットし，呼び出すと，インストール済みのDEVICE_IDがセットされます．戻り値がゼロ以外ならばエラーです．

● Close_Driver(void);

この関数は開いているデバイスをすべて閉じます．戻り値がゼロ以外ならばエラーです．

では，この関数を用いたテスト・プログラムをリスト1に示します．インストールされているデバイスをオープンしてクローズするだけです．

結果としてデバイスIDを表示しています．

コンパイルは次のように行います．

```
bcc32 test100.c port_2k.lib
```

実行結果は次のようになります．

```
C:¥Documents and Settings¥test¥デスクトップ¥
                                  test>test100
DEVICE_IDは = 264です
```

リスト1 test100.c

```
#include <stdio.h>
#include <conio.h>
#include <windows.h>

#include "port_2k.h"
#include "adm686z.h"


void main(void)
{
    int  i, j, brd_num, rv, device_id[8], index[8];
    BYTE data;
    DWORD start_time;
    BOOL quit, err;

    for (i = 0; i <= 7; i++)
    {
        device_id[i] = -1;
        index[i] = -1;
    }

    brd_num = 0;
    rv = Open_Driver(device_id);
    if (rv != 0)
    {
        printf("Open_Driverでエラー(%d)が発生しました。¥n", rv);
        exit (-1);
    }

    for (i = 0; i <= 7; i++)
        if (DEVICE_ID ==device_id[i])
            index[brd_num++] = i;

    printf("DEVICE_ID= %dです¥n", DEVICE_ID);
    rv = Close_Driver();
    if (rv != 0)
    {
        printf("Close_Driverでエラー(%d)が発生しました。¥n", rv);
    }
    exit(-1);
}
```

```
C:¥Documents and Settings¥test¥デスクトップ¥
                                       test>
```

このport_2k.libを使用したソースは，製品に付属しているCD-ROMに入っています．TD686W2.C，TD686ZW2.Cがそれです．

このソースを参照すれば簡単なアプリケーションを作ることができます．

もちろんGUIを使ったものでもヘッダH686Z.Hを参照して，本来のライブラリ関数を使うことで，LaBDA Q-A Qのようなアプリケーションも作ることもできます．

WindowsのGUI処理が苦手であればVBも使用することが可能です．また，Delphiにも対応しているので，慣れている人ならば，利用してみるのもよいでしょう．

このように高性能なA-D変換ボードを使用すると，プログラムも簡単に書けて，リアルタイムOSを使用する必要性も薄れます．機会があれば，もっと手軽に利用できるA-D変換ボードを使用した計測について書いてみようと考えています．

きし・てつお

Chapter 2

やりなおし物理学実験

ノイズと測定誤差の補正方法

藤井 研一

アナログ，ディジタル機器にかかわらず，ノイズを減らすために膨大な努力が繰り返し行われている．その蓄積に基づいて，ノイズ逓減技術に関する本が数多く出版されている．一方，このようなノイズにより本来の値から測定値がずれてしまう，いわゆる誤差の存在を前提として，誤差自身の性質を考えてこれをどのように取り扱ったらよいかについて述べる．そのうえで精度の良い測定はどうすればよいのかを理解するために，誤差論について解説する．（筆者）

実際に計測に関わって，ノイズに悩まなかった人はいないだろう．

測定において意識的に誤差の話が出てくるのは，大学などでの物理の実験ではないだろうか．

確かに物理の実験現場では，精密測定が求められており，探り出したい情報をノイズの海の中からいかに拾い出すかがつねに付きまとう．このような精密測定の基礎を与えるため，大学などでは初年度の実験から有効数字や誤差などについてしつこくいわれたことを思い出すのではないだろうか．あまりにもうるさくいわれすぎたせいか，実際の業務に就くと誤差の問題はなんとなく処理して済ましてしまうことが多いという読者もいるだろう．

あらためて有効数字とは何だと聞かれたとき，すらすらと答えることができるほど自信があるだろうか．ここではもう一度，学生のころに戻り，物理の実験を行うつもりで，誤差の話を復習してみよう．

抵抗や電子デバイスの特性を調べるために，温度変化の測定を行うことも多い，このとき，通常の温度域では手軽にかつ高精度に温度を測定できるため，熱電対を使うことがしばしばある．熱電対の示す起電力をディジタル・マルチメータ（DMM）で表示するとともに，A-D変換してデータをコンピュータに取り込んだ経験のある方も多いのではないだろうか（図1）．この熱電対を例として，測定の誤差評価を考えてみる．



図1　熱電対を用いた温度測定のようす

また，どうすれば誤差を小さくすることができるかについても考えてみる．

熱電対による温度測定

熱電対は異なる2種類の金属を溶接して接続したもので，異なる金属の間に発生する起電力の大きさから温度を測定するものである．このために二つの接点が必要で，一方の接点は基準となる温度に保つ必要がある．普通はこの端子（基準接点）を氷水の中につけ0度に保つ．そしてもう一方の接点（測定接点）に発生する起電力を測定すれば良い．

熱電対としてはさまざまな金属が使われるが，もっとも広く使われているものには白金–ロジウム，クロメル–アルメル，銅–コンスタンタンといったものがある．

これらの中から測定温度などの条件に合わせて適当なものを選択し，利用する．どれを用いても手軽にかつ精度良く温度を測定できるため，非常によく利用されている．

発生する熱起電力は熱電対の種類によっても異なるが，数mV程度の値となる．

図2にクロメル–アルメル熱電対の温度と発生する起電力の関係を示す．通常，適切な温度域で良い直線性を示す．理科年表などでこの起電力の基準値は参照できるが，実際の熱電対の起電力は必ずしもこの値どおりにはならない．そのため精密な温度測定を行う場合，用いる熱電対の較正を行う必要がある．図のクロメル–アルメル熱電対で測る場合，精度良く温度を測定するためにはμV程度までは精密に較正しておく必要がある．したがって，ノイズの多い環境で測定すれば，すぐに数℃以上の温度のあいまいさが生じてしまう．



誤差と有効数字

● ディジタルとアナログの先入観

よくある思い込みに，アナログだと誤差があるがディジタルには誤差がないというものがあるが，これは誤解である．

ディジタルに変換されたDMMでは値が数字で示されるのに対し，アナログのテスタでは針が指し示す値を数値として読むためには注意と時間が必要なため，このような思い込みが生じたと思われる．

しかし，もともとのデータはアナログであり，これがA-D変換されて表示されることを思えばこれは明らかなまちがいである．もちろん，A-D変換時に丸め誤差が生じる場合もあるが，それ以前に実は何桁目まで信じられるかがよくわからない場合が多い．

確かに5 1/2桁のディジタル・マルチメータでは5桁の数値が表示される．ただ，サンプリング&ホールドしてみると，あるときには，たとえば0.7984mVと表示されるだろうが，適当な間隔の後にもう一度ホールドしてみると0.7821mVとなってしまうかもしれない．

この場合，表示している桁はすべて有効ではない．何桁目まで正しいのかを明らかにするためには繰り返し測定を行うしかない．

これから述べる誤差の説明では，まず読み取りの誤差を考えるが，ディジタル表示の場合，このように誤差の生じる桁がどこにあり，どのくらい変化しているのを読み取るのが実は困難な場合がしばしばある．この意味で，有効な測定値を考えるうえでは，アナログ測定器のほうが理解しやすい場合が多い．

● 誤差が生じる要因

どのような測定器を用いて測定しても測定できる桁は有限である．また，どれほど細心の注意を払って測定しても，測定のたびに最小の桁まで同じ値になることはない．

通常，アナログの測定器では，読み取れる最小の桁は最小目盛りよりも小さな値を目分量で読むこととなる．適切なレンジを選択して値を読み取っても，読み取れた値は測定のたびに少しずつ異なっている．

異なっているが，どの桁まで読み取れて，どの桁にあいまいさがあるかは明らかである．

ところで，ここで測定を試みている量の本当の値(真値)はただ一つであるはずなので，毎回この真値とは異なった値が測定されていることとなる．

この真値と測定値のずれが誤差となるのだが，誤差を生じる原因は，たいへん複雑である．

まず，最小の値を読み取るときに目分量になるため，あいまいさが入ってくるが，ここには個人の読み取りの癖が入ってくる余地がある．たとえば，アナログのテスタで値を読む場合，値を指す針を真上から見ずにつねに右からみる癖を持っている人では，値はつねに小さめに読み取られることとなる．

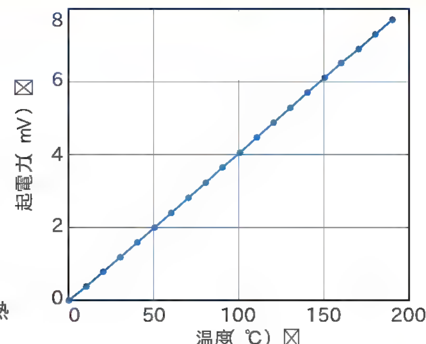


図2
クロメル-アルメル熱電対の標準起電力

一方，測定器の調整が十分でない場合には，つねに真値と比べて値が大きく(あるいは小さく)表示される可能性もある．前者は訓練によりある程度は補正できるであろうし，後者は測定の調整を十分に行うことで小さくすることができる．

しかし，誤差にはこれ以外にランダムな要素により生じる偶然誤差と呼ばれるものも存在する．

おおざっぱにいえば，このような原因で誤差が生じるが，その特徴を含めてまとめてみると次のようになる．

● 誤差の種類

- 組織誤差：測定器の目盛りのずれや調整不良によるずれが原因の誤差．装置の調整で取り除くことが可能．ただし，通常の測定では較正を一から十分に行うことは困難なため，機器の指定の較正条件をチェックして避けるようにする．
- 個人誤差：アナログの測定器において目盛りを読み取るとき，目盛りの示す値の間を目分量で読み取る．このためおよそ最小目盛りの1/10の桁の値が読み取れる．ただし，目分量で読み取るために個人の「癖」によるバイアスがかかる．この「癖」による誤差．
- 偶然誤差：上記の組織誤差，個人誤差以外の要素により現れる誤差が存在し，それらを総称して偶然誤差と呼ぶ．たとえば測定値自身の物理的な揺らぎなどにより，値が幅を持ってしまうような場合．

組織誤差，個人誤差を「合わせ系統誤差」と呼ぶ．これらは測定値をある方向にずらすように現れる．組織誤差は機器の調整によって，個人誤差は訓練による「癖」の除去によってある程度の修正は可能であるが，一般に取り扱いは難しい．

一方，偶然誤差は通常，多数回の測定による統計的な取り扱いにより縮小することが可能である．ここでは偶然誤差を中心として話を進めることとする．

● 誤差の表記と有効数字のとらえ方

実際の測定を考えてみよう．まずアナログの測定器での測定を考えてみる．たとえば上記の熱電対の測定で，得られた熱起電力の値が V = 0.783mVと読み取れた場合，最後の桁の数字3には読み取りのあいまいさが入っていることになる．

このようにあいまいさが最初に現れる桁までの数値を表示することにより，その数値の確からしさが表記から読み取れるこ

となる．この信用できる数値を有効数字と呼ぶ．

この場合は3桁ということになるが，最小の桁には当然あいまいさ（誤差）が含まれることとなる．測定値xはこのあいまいさの大きさ程度は幅を持つことになる．

一般に，誤差δxは正の値と考え，$x \pm \delta x$と表記する．これは測定値が$x - \delta x$から$x + \delta x$の範囲にあることを示す．それでは最小の桁にどの程度のあいまいさが含まれているのだろうか．1回の測定ではそのあいまいさははっきりとは指摘できない．このため，最小の桁に1程度のあいまいさが含まれているとして誤差δVを考えると，

$$V \pm \delta V = 0.783 \pm 0.001 \cdots\cdots (1)$$

のような表記をするのが自然である．ここで誤差として採用した±0.001は正確ではないが，測定器の目盛りの幅などからおよその大きさを見積もって表す．

また，誤差はその値自体が重要ではなく，測定値に対してどのくらいのあいまいさがあるかということのほうが意味をもつ場合が多い．この場合，次のように相対誤差として測定値の何%の誤差かを示すこともしばしば行われる．

$$\frac{\delta V}{V} \times 100 (\%) \cdots\cdots (2)$$

繰り返し測定での誤差

● 繰り返し測定での最確値を求めると

それでは同じ測定を一定条件の下（今の場合は一定温度の下）で繰り返し行うとどうなるのだろうか．まず，測定器は十分に調整し，データ値を読み取る人間もきちんと訓練を行い，系統誤差はないものとして話を進めよう．

そのうえで5回の繰り返し測定を行って表1のような熱起電力V（mV）の値を得たとしよう．

この場合，どの桁までの値が信用できるかはもう少しはっきりと表すことができる．まずもっとも確からしい値（最確値）としては，この五つの値の平均値V_{av}を用いれば良い．

$$V_{av} = \frac{1}{5}\sum_i V_i = \frac{1}{5}(0.788 + 0.793 + 0.784 + 0.790 + 0.791) = 0.7892 \cdots\cdots (3)$$

また，得られたもっとも大きな値ともっとも小さな値を使って誤差δVを見積もるならば，あいまいさを，

$$2\delta V = 0.793 - 0.784 = 0.009 \cdots\cdots (4)$$

と考えることができる．このため前節と同じような表記を行うならば，

$$V_{av} \pm \delta V = 0.789 \pm 0.005 \cdots\cdots (5)$$

のように書き表すことができる．これを誤差と考えるのはかなり荒っぽい話なのでもう少し確からしい誤差を考えてみよう．

いま，N回の測定値x_i（$i = 1, \cdots, N$）の最確値として平均値$\bar{x}$を取った場合，各測定値が平均値からどのくらいずれているかという残差（偏差）$x_i - \bar{x}$を測定値すべてに対し平均することでより確からしい誤差とすることができそうに思われる．

しかし平均値の定義より，このままではつねにゼロとなってしまうので誤差を考えるうえで，次のように平均値からのずれの自乗の平均値を求めることにする．

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2 \cdots\cdots (6)$$

この量を分散と呼ぶ．さらに分散の平方根をとると元の量と同じ次元（単位）を持つのでつごうが良い．これを標準偏差σと呼ぶ．

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2} \cdots\cdots (7)$$

偶然誤差のみ存在する場合，測定値に対してこの標準偏差をそのまま誤差と考えて良い．上の起電力の例に対して，誤差を求め直し，最確値を書き表すと，

$$V_{av} \pm \delta V = V_{av} \pm +\sigma = 0.789 \pm 0.003 \cdots\cdots (8)$$

となる．いずれにしろ，この測定では測定値の有効数字は3桁となる．

● 繰り返し測定における偶然誤差と系統誤差

ここでさらに繰り返し測定を行った場合，測定点のばらつきぐあいが誤差の種類によってどうなるかを考えてみよう（図3，図4）．

今，真の値は図の直線上にあると考える．図3は偶然誤差が小さい場合の測定点のばらつきを示している．円Aで囲まれた測定点は系統誤差も小さい場合を示しているが円Bで囲まれた測定点には一定の系統誤差が含まれている．

このため，Aでは測定点の平均値は真の値程度になるが，Bの場合には測定点の平均を取っても真の値からずれた値となる．

偶然誤差が大きい場合を図4に示す．今，真の値を直線として示しているので，それからのずれぐあいを直感的に理解できるが，現実には真の値はこの直線のように明らかではない．

このため系統誤差が大きい場合には，同じような測定値のばらつきが真値からずれた値を中心として生じることになる．この場合，真の値を得ることは困難となる．真の値を得るためには偶然誤差を小さくすることはもちろんであるが，系統誤差が存在する限り真の値を得ることができないこと，したがって系統誤差を減らす努力，たとえば測定器の十分な較正は必要条件であることがわかると思う．

ただ，一般に系統誤差の原因は単純ではないため，特定して取り除くことは困難な場合が多いことにも注意しよう．

表1　繰り返し測定した結果

1	2	3	4	5
0.788	0.793	0.784	0.790	0.791

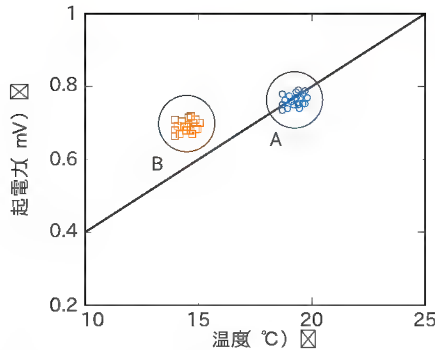


図3　偶然誤差が小さい場合の測定結果

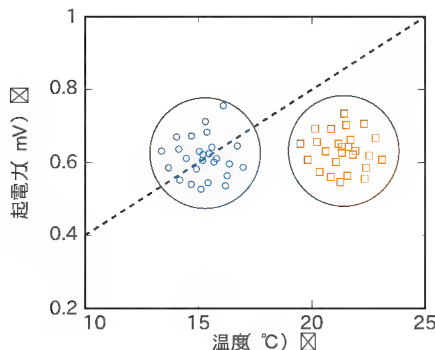


図4　偶然誤差が大きい場合の測定結果

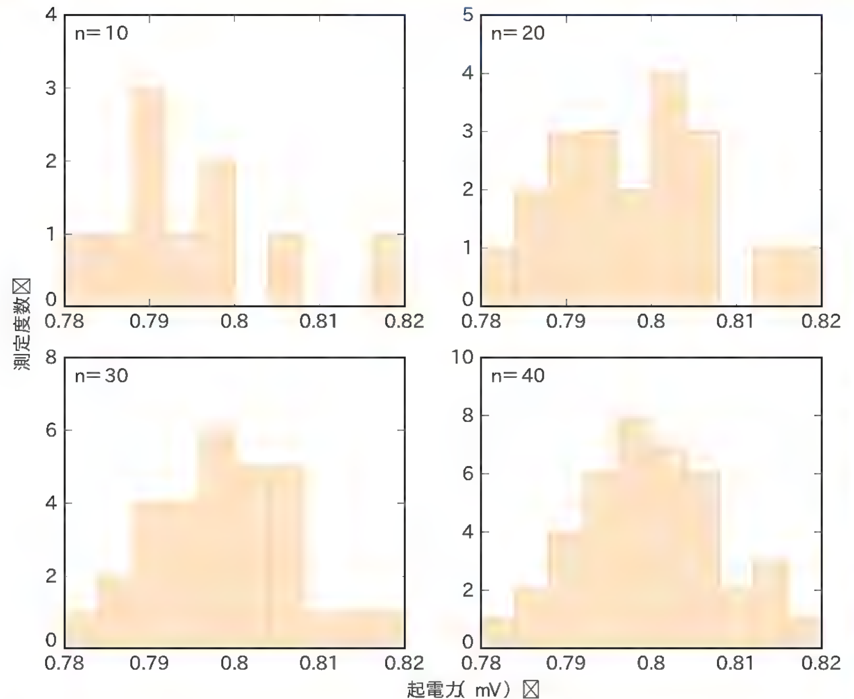


図5　40回の繰り返し測定をプロットした結果(ヒストグラム)

統計を考慮した繰り返し測定

次にDMMを用いて一定条件の下に40回の繰り返し回数で熱電対の値を測定した場合を考えてみよう．測定値は表2のように5桁の値が得られたとする．

前節と同様にして平均値V_{av}は次のようになる．

$$V_{av} = \frac{1}{40}\sum_{i=1}^{40} V_i = 0.80019 \quad \cdots\cdots (9)$$

誤差として再び標準偏差を考えるならば，

$$\sigma = \sqrt{\frac{1}{40}\sum_{i=1}^{40} V_i^2} = 0.80080 \quad \cdots\cdots (10)$$

となるので，この場合の起電力としては0.800± 0.008(mV)となる．有効数字は誤差によって決まってしまうので，必ずしも0.80019のように測定値と同じ桁になるというわけではない．

上記のように温度の測定を同一条件で繰り返し行うと，得られる値はある範囲内にばらつく．観測される起電力を横軸にとり適当な区間(グラフでは0.04刻み)に分け，縦軸に測定度数値をとる．繰り返し測定の結果をこのグラフ上にプロットすると，ヒストグラム(度数分布)と呼ばれる図形が得られる．

たとえば，表2の測定値の最初の10個，20個，30個，40個のヒストグラムを作ってみよう(図5)．

繰り返し測定の回数が少ない場合，ヒストグラムの規則性ははっきりしない．しかし繰り返し回数を増すにつれて規則的な

表2　40回の繰り返し測定をした結果

	1	2	3	4	5
1	0.78298	0.79891	0.81699	0.78995	0.79094
2	0.80591	0.78792	0.79291	0.79798	0.78891
3	0.80791	0.81297	0.79290	0.80200	0.80398
4	0.79394	0.80792	0.80297	0.78794	0.80099
5	0.80695	0.79794	0.80893	0.79891	0.79490
6	0.78990	0.80698	0.79697	0.79896	0.80194
7	0.81391	0.81191	0.79898	0.79592	0.80490
8	0.80197	0.79498	0.80292	0.79998	0.81291

形状に近づいていく．その特徴を述べると，

1) 中心値でもっとも測定度数が高くなる
2) 中心値から値が離れるほど測定度数が小さくなる
3) 測定値の値のばらつき方は中心値の前後で対称となっている

となることが予測できるだろう．

これは誤差として偶然誤差のみ存在する場合の特徴となっている．繰り返し測定回数を無限に多くした極限ではこのヒストグラムは上記の特徴をもつなめらかな関数形を示すことになる．これを極限分布と呼ぶ．極限分布関数は偶然誤差のみを考える場合，正規分布関数(ガウス関数)となり，測定値xに対しては次のような形で与えられる．

$$P(x) = \frac{1}{\sqrt{2\pi\sigma}}\exp\left[\frac{-(x-x_0)^2}{2\sigma^2}\right] \quad \cdots\cdots (11)$$

ここでx_0は真の値を示す．ところで，この真の値は無限回の極限としての正規分布関数がわかって初めて出てくる値である．

現実にはこれは不可能で，有限回の測定で考えねばならない．

News Flash ―― NECと日興通信，RFIDを活用した書類収納管理システムの開発に着手
NECと日興通信(株)は，RFIDを活用し，大量の書類の所在確認や入出庫管理などを効率化する「書類収納管理システム」を共同で開発すると発表した．同システムは名古屋銀行への導入が決定している．2005年4月から稼動する予定．

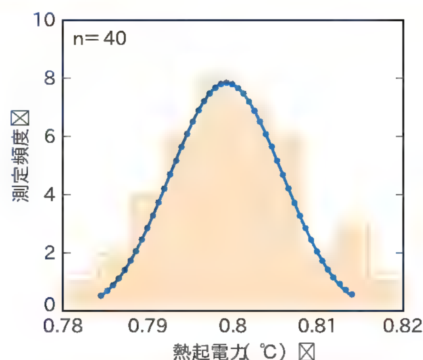


図6
ヒストグラムに正規分布関数を重ねる

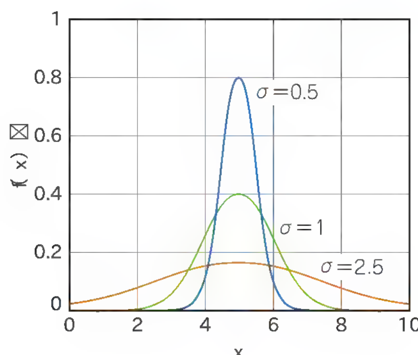


図7
同じ中心値で異なる標準偏差をもつ関数

そこで真の値に代わる最確値をどうするかという問題になる．後で述べるように平均値を最確値として用いて良いことがわかる．

したがって，上の式は平均値，

$$\bar{x}=\frac{1}{N}\sum_{i}^{N}x_i \quad \cdots\cdots (12)$$

を用いて，

$$P(x)=\frac{1}{\sqrt{2\pi}\sigma}\exp\left[\frac{-(x-\bar{x})^2}{2\sigma^2}\right] \quad \cdots\cdots (13)$$

で与えられることになる．

図6に $n=40$ のヒストグラム上にこの正規分布関数を重ねて示した．測定回数が増すにつれて正規分布関数に近づくことが理解できるものと思われる．正規分布関数の形は，σによって大きく変化する．図7に同じ中心値で異なる三つの標準偏差 $\sigma=0.5$，1，2.5をもつ関数のグラフを示した．

σの値が小さいほど鋭く，大きくなるにつれ広がった関数形を与えることがわかる．したがって，σはこの関数の幅を決める量と考えることができる．式(13)において指数関数の前に因子 $1/\sqrt{2\pi}\sigma$ がついている．この関数の特徴に測定値の全範囲として－∞から∞を取って積分すると，

$$\int_{-\infty}^{\infty}P(x)dx=1 \quad \cdots\cdots (14)$$

のように規格化されていることがわかる．

$P(x)$ は，値 x が出現する確率を与える量であるから，この分布関数に対する平均値 x は次のようになることは容易にわかる．

$$\bar{x}=\int_{-\infty}^{\infty}xP(x)dx=x_0 \quad \cdots\cdots (15)$$

つまり，極限の分布としての正規分布では平均値 x は真の値 x_0 となる．同様に真の値 x_0 からのずれ $x-x_0$ の自乗の平均値を計算すると，

$$\int_{-\infty}^{\infty}(x-x_0)^2P(x)dx=\sigma^2 \quad \cdots\cdots (16)$$

となり，σは式(7)の N を無限大の極限に取ったときの標準偏差と同じになる．これより，有限回の測定での誤差は式(7)で与えられる標準偏差と考えて良いことがわかる．

誤差伝播則

● DMMによる測定

DMMを用いて熱電対の電圧 V_s を測ることに戻ろう．

信号源の抵抗を R_s，DMMの入力インピーダンスを r_{in} とすると，電圧値 v_{disp} から V_s は次のような式で与えられる．

$$V_s=\left(1+\frac{R_s}{r_{in}}\right)v_{disp} \quad \cdots\cdots (17)$$

当然，V_s をできるだけ正確に測るためには入力インピーダンスは大きくならなければならないが，この存在のために，表示値は信号源の電圧と同じにはならない．

さらに入力インピーダンスの値や信号源の抵抗を正確に知っておくことが電圧値 V_s を得るために必要になる．

ところが，これらの値も測定することでしか知ることができないため誤差を持つことになる．では，これらの誤差が求めようとしている電圧値の誤差にどのような影響を与えるのだろうか．

この場合，次に述べる誤差伝播を考える必要が出てくる．

● 誤差伝播則を求める起電力の誤差

一般的に，求めたい量がある関数 $y=f(x_1, x_2, x_3, \cdots)$ により表されるとき，その(平均)誤差 σ_y は変数 x_i の誤差から次のように求めることができる．

$$\sigma_y=\sqrt{\left(\frac{\partial f}{\partial x_1}\sigma_1\right)^2+\left(\frac{\partial f}{\partial x_2}\sigma_2\right)^2+\left(\frac{\partial f}{\partial x_3}\sigma_3\right)^2+\cdots} \quad \cdots\cdots (18)$$

この式を用いるならば，たとえば長さ ℓ と半径 a から与えられる円柱の体積 $V=\pi\ell a^2$ において長さと半径の誤差が σ_ℓ，σ_a とわかっているならば，体積の誤差 σ_V は次のように与えられる．

$$\sigma_V=\sqrt{(\pi a^2\sigma_\ell)^2+(2\pi a\ell\sigma_a)^2} \quad \cdots\cdots (19)$$

熱電対のDMMでの測定では，起電力は式(17)のように表示電圧値，入力インピーダンス，測定源の抵抗値の関数として表される．これらの量とそれぞれの誤差 σ_v，σ_R，σ_r を用いて誤差伝播則を求める起電力の誤差 σ_V が求められることになる．

$$\sigma_V=\sqrt{\left(\frac{v_{disp}}{r_{in}}\sigma_R\right)^2+\left(\frac{R_s v_{disp}}{r_{in}^2}\sigma_r\right)^2+\left(\frac{R_s}{r_{in}}\sigma_v\right)^2} \quad \cdots\cdots (20)$$



繰り返し測定の意味？

● 誤差の統計的扱い

誤差として標準偏差を取れば良いことはすでに述べた．式(7)の定義では測定回数$N = 1$でも値を持つことになる．繰り返し測定における値のばらつきから誤差を読みとろうという考え方からすれば，無理があることになる．

そこで，式(7)の代わりに分母の因子Nを$N-1$として定義して用いることが多い．すなわち，

$$\sigma = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})^2} \quad \cdots\cdots (21)$$

である．こうすると，測定回数が1回の場合，分母，分子ともにゼロとなり，値が定まらないことになる．

通常の測定値では$N = 10$回程度のものであろうから，式(7)と式(21)の標準偏差の定義式の違いは0.333に対して0.316となり，約5%程度の誤差の違いとなる．したがって，両式は本質的な違いを与えないと考えられる．

繰り返し測定に対する値として考えるならば，1回の測定では値が不定となる後者を誤差として選んだほうがつごうが良いことになる．

正規分布関数の標準偏差がσで与えられることは，1回の測定で中心値x_0の前後$\pm\sigma$の中に測定値が入ってくる確率が68.26%となることを意味している．さらに$x_0 \pm 2\sigma$の範囲の中には95.46%の確率で測定値が入ることとなる．これはどの1回の測定に対しても成り立つ．熱電対は基準値があるが，個々の熱電対には温度と起電力との関係にばらつきがある．

つまり，ある温度Tでの起電力$V(T)$は一つ一つ異なっていると思われる．このばらつきを熱電対1個1個について調べるのはたいへんな労力となるため，ほぼ不可能である．

ところで，熱電対が工場などで作られる条件は一定に保たれ，同じと考えられるので，ある温度での起電力の値$V(T)$に対する誤差σ_Vはどれもほぼ同じと考えられる．

そこで，作られた熱電対から一つを選び出し，温度Tで数回起電力を測るならば，最適値としての平均値と誤差(標準偏差)σ_Vが求められる．ここで求められた誤差は，同様に測定する限り，どの熱電対でも同じと考えることができる．

したがって，それぞれの熱電対で1回起電力を測定すれば，その値は68%の確率で真の値の前後$\pm\sigma V$の範囲に入ってくることとなる．

● 平均値の誤差(標準偏差)

今，同じ測定を無限に繰り返すことを，次のようにN回の測定を1セットとして考え直してみよう．

つまり，ある量の平均値を求めるためにN回測定を行う．この平均値を求める試みを何度も繰り返すと考えてみる．すると平均値xは，

$$\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i \quad \cdots\cdots (22)$$

となるが，これは平均値xが独立なN個の変数$x_1, x_2, x_3, \cdots, x_N$の関数となっていると考えることができる．

$$x = f(x_1, x_2, \cdots, x_N) \quad \cdots\cdots (23)$$

また，それぞれの変数x_i($i = 1, 2, \cdots, N$)は，さまざまな値を取り得るが，どれも同じ正規分布関数に従うと考えられる．このため，それぞれの変数値x_iの誤差σ_iは同じ値σとなると考えられる．したがって誤差伝播の式から，

$$\frac{\partial f}{\partial x_i} = \frac{1}{N} \quad \cdots\cdots (24)$$

となるため，平均値xの誤差σ_xは次のようになる．

$$\sigma_x = \sqrt{\left(\frac{1}{N}\sigma_1\right)^2 + \left(\frac{1}{N}\sigma_2\right)^2 + \cdots + \left(\frac{1}{N}\sigma_N\right)^2} = \frac{\sigma}{\sqrt{N}} \quad \cdots\cdots (25)$$

式(21)を用いて平均値の標準偏差は次の式で与えられることになる．

$$\sigma_x = \sqrt{\frac{1}{N(N-1)}\sum_{i=1}^{N}(x_i - \bar{x})^2} \quad \cdots\cdots (26)$$

したがって，先で例とした起電力の測定値に対して，平均値に対する標準偏差を求めると$\sigma_x = 0.0012$となることがわかる．これは40回の繰り返しにより16%まで小さくなっていることになる．

しかしながら測定回数Nに対し平均値の誤差は$1/\sqrt{N}$で小さくなっていく．10回の測定回数を100回に増やしても平均値の誤差は測定値の誤差の0.316から0.1に減るだけということを意味している．測定回数を増やすことは平均値の誤差を減らすことに有効であるが，測定回数がある程度増えてしまうと誤差向上のためにそれ以上回数を増やしても効率はあまり良くないということになる．

● 最小自乗法で最適な関数を求める

測定を繰り返すと測定値の分布は(11)で与えられる分布関数に近づくことを述べたが，この関数は最確値で最大値をとる．今N回の測定を繰り返し，($x_1, x_2, \cdots x_N$)という一組の測定値が得られたとする．この測定値はそれぞれ互いに独立であると考えられるので，誤差はどれもσで与えられる．すべての測定値は，

$$P(x_i) = \frac{1}{\sqrt{2\pi}\sigma}\exp\left[\frac{-(x_i - x_0)^2}{2\sigma^2}\right]$$

という確率で与えられることになる．したがって($x_1, x_2, \cdots, x_N$)という測定値の組が得られる確率は，それぞれの得られる確率の積で与えられることとなる．

$$P(x_1, x_2, \cdots, x_n) = \prod_{i=1}^{n}\frac{1}{\sqrt{2\pi}\sigma}\exp\left[\frac{-(x_i - x_0)^2}{2\sigma^2}\right] = \left(\frac{1}{\sqrt{2\pi}\sigma}\right)^n \exp\left[\frac{-1}{2\sigma^2}\sum_{i=1}^{n}(x_i - x_0)^2\right] \quad \cdots\cdots (27)$$

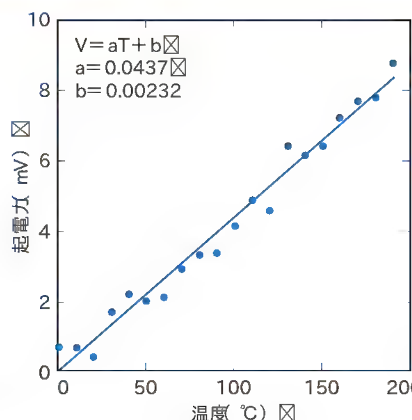


図8
熱伝対の起電力を温度の関数として測定

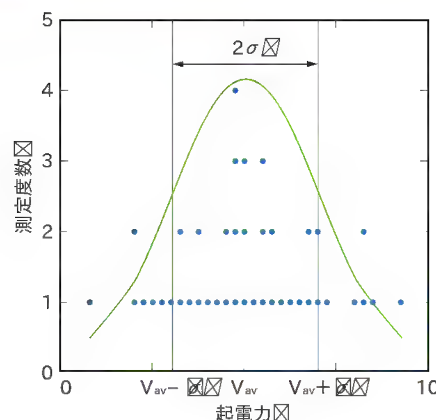


図9
熱電対の測定値の測定頻度と正規分布関数

上式の真値x_0を最確値μ(平均値$\bar{x}$として考えてきた値)で置き換えたものを尤度(ゆうど)関数と呼ぶ．式(27)の$P(x_i)$は最確値で最大となるから，尤度関数$P(x_1, x_2, \cdots x_N)$もまた最確値μにおいて最大値をとることとなる．したがって，μにおいて，

$$\frac{\partial P(x_1 \boxtimes \cdot \boxtimes)}{\partial \mu} = 0 \cdots\cdots (28)$$

を満たすこととなる．これは指数関数のべきを最小にすることと同値である．したがって，

$$\frac{1}{\partial \mu} = \left(\sum_{i=1}^{N} (x_1 \cdots \boxtimes)^2 \right) = 0 \cdots\cdots (29)$$

となる．

さらに同様に考えて誤差の最確値は次の式で与えられる．

$$\frac{\partial P(x_1 \boxtimes \cdot \boxtimes)}{\partial \sigma} = 0 \cdots\cdots (30)$$

このように，測定値が正規分布に従う場合，最確値を求める方法は上で述べたことから明らかなように測定値と最確値の残差を最小にすることで得られることとなる．これを最小自乗法と呼ぶ．

図8に適当な熱伝対の起電力を温度の関数として測定したものを示す．最初に示したように熱電対の起電力は温度に対し良い精度で直線的に変化する．この測定値に対し最小自乗法により最確値との残差の自乗の和を最小にするようにして求めた直線を示している．

関数形が明らかな場合，最小自乗法により直線以外の関数であっても最適な関数を求めることができる．

● カイ自乗検定で測定値の分布を調べる

測定した値を統計的に考えてきたが，その取り扱いの元となっている分布関数が本当に正規分布関数であるかどうかは実は定かではない，分布関数としては二項分布，ポアソン分布，ローレンツ分布なども考えられる．ここではこの分布関数に対する仮定が正しいかどうかを考えてみよう．正規分布関数に従うならば，中心となる最確値±誤差の範囲には68%のデータが入ってくることになる．

したがって，期待される観測回数(期待度数)n_{exp}と観測度数n_mのずれ$n_m - n_{exp}$は仮定した極限分布が正しく，観測回数が大きければ十分小さいものと期待できる．逆に，予想に反してずれが大きければ，仮定した極限分布関数が誤っているということになる．この大小を判断するものとして次のカイ自乗という量を考える．

$$\chi^2 = \sum_{i=1}^{N} \frac{\left(n_m - n_{exp}\right)^2}{n_{exp}} \cdots\cdots (31)$$

ここで$\chi = 0$ならば測定データの分布は仮定している分布関数に完全に一致していることになる．実際には上式の各項の大きさはおよそ1程度と考えられるので，測定回数Nと比較して，

$$\chi^2 \leqq N \cdots\cdots (32)$$

となるなら仮定した分布は正しいことになる．逆に，

$$\chi^2 \geqq N \cdots\cdots (33)$$

ならば期待度数と観測度数の差は有意であり，実際の分布は仮定したものとは異なると考えたほうが良いこととなる．

図9に先に示した熱電対の測定値の測定頻度とこれより求めた正規分布関数を示した．中心となる平均値±標準偏差の間に27個の値が入ってくることがわかる．これは40個の測定値の67.5%となっており仮定した分布は正しいと考えることができる．

おわりに

誤差は測定には避けて通れないものである．さまざまなくふうにより測定時のノイズを減らすことはできるし，その努力を惜しむべきではない．そうであっても最終的に偶然誤差は残ってしまう．せっかくのノイズ逓減の努力が，測定値を正しく扱わなかったばかりにかえって誤差を過大評価してしまってはもったいない話である．偶然誤差の統計的扱いは決して難しくはない．一方で系統誤差に対する一般的な処理方法というものはない．誤差の性質を正しく理解し付き合っていくことは計測に携わるうえでの必須条件である．この記事が正しい誤差の理解へ向かうきっかけとなることを切に希望する．

ふじい・けんいち　大阪大学理学部物理学科

Chapter 3

雑音に埋もれた少ないデータから統計処理やスペクトル解析を精度よく行う

統計処理とスペクトル解析の技

尾知 博

本章では，少ないデータ数や雑音を含むデータでも，なるべく精度よく種々の統計量や周波数スペクトルを求める方法や雑音を抑圧する方法について，ディジタル信号処理技術の応用であるマルチレート信号処理やウェーブレット変換を用いて検討してみることにする．実用的なMATLABやO-Matrixで書かれた演習ファイルもダウンロードできる（http://www.cqpub.co.jp/interface/）ので，読者はみずから計算機シミュレーションを行いながら確認してもらいたい．（筆者）

ある信号に対して，平均，分散，標準偏差，相関といった統計量の解析，あるいは周波数スペクトル解析を行う場合，データが欠損あるいは十分な測定時間がなく，少ないサンプル/データ数で処理を行う必要がある場合がある．また，測定環境やセンサによっては，雑音を含んだ信号に対するデータ処理を行う必要がある．

本章で解説する事項

データの統計処理や周波数スペクトル解析を実際に行う場合，何が問題となるのであろうか．それは，

1) 少ない(短い)データ数
2) 雑音を含んだデータ

しか得られない場合でも，なるべく精度よくデータ処理を行う必要があるときであろう．本章では，こうしたデータ処理の問題に対するディジタル信号処理技術をいくつか紹介する．

具体的には，まずスプライン補間やディジタル・フィルタを用いたデータの補間方法を説明し，少ないデータ数でも精度良く統計量の推定が可能か検討してみる．次に，少ないデータで周波数スペクトル解析を行う際に，解析精度すなわち周波数分解能を上げるいくつかのテクニックを紹介する．そして，S/N比が劣悪な環境，すなわち雑音に埋もれた信号から，その信号の性質(周期性，不連続性)を調べたり，雑音を抑圧できるウェーブレット変換について述べる．

本章は，統計学や信号処理の復習や解説を行うために書いたのではないので，これらの分野の内容を網羅しているわけではない．解説では極力数式を使わずに物理的な説明を行い，直感的な理解ができるよう試みるが，期待値，相関関数，不偏・一致推定量，フーリエ変換，FFT，z変換…などの数学的事項は，必要に応じて教科書や参考文献も見ていただきたい．

計算には，MATLABのほかに補間など関数近似や行列計算に関するコマンドが豊富なO-Matrixも利用する．各演習タイトルの右側にファイル名を示しているので，読者はダウンロードして実際にシミュレーションを試していただきたい．興味が湧くはずである．

少ないデータ数で統計処理をする

本項では，平均値や相関係数などの時間領域の統計量を，少ないサンプル・データ数で精度よく推定する方法について述べる．まず，基本的な最小自乗近似やスプライン補間などの関数近似によるデータ補間手法を説明する．ついでインターポレーション，すなわちマルチレート信号処理技術を用いた信号処理論的に“正確”な補間手法についても言及し，株価予測に応用してみることにする．

● 統計量とデータ数

最初に平均値と自己相関値を例にとり，どの程度のデータ数が統計量の推定に必要なのか，以下の例題で見てみることにしよう．

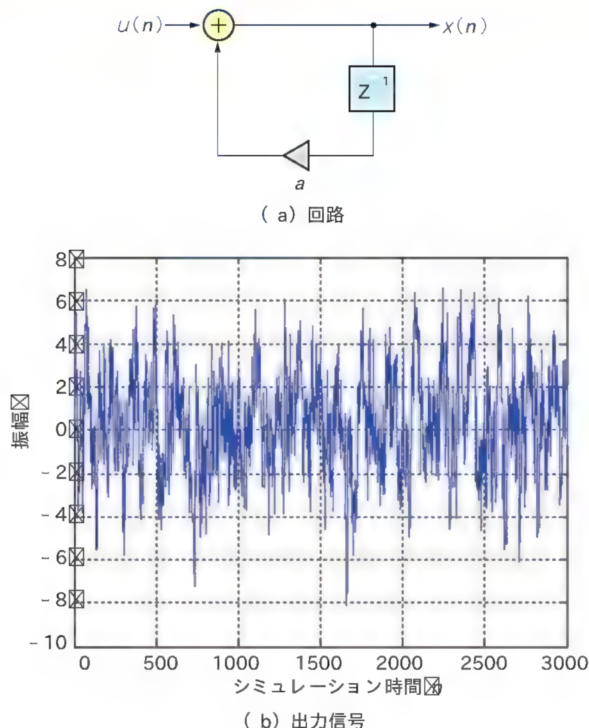
演習1 1次AR過程の平均値と自己相関関数　ex2_1_5.m

図1は，1次AR(Auto-Regressive：自己回帰)過程をディジタル信号処理的に発生させる回路である．入力$i(n)$は，平均ゼロ，分散σ^2の白色雑音である．nはサンプル時間を表す．この1次AR過程$x(n)$の平均値と分散および相関関数について，横軸をサンプル数にして測定せよ．ただし，$a=0.9$，$\sigma^2=1$とする．

解

結果を図2に示す．このように，一致推定量に近似するには平均値および分散値ともに1500サンプル程度必要であることがわかる．

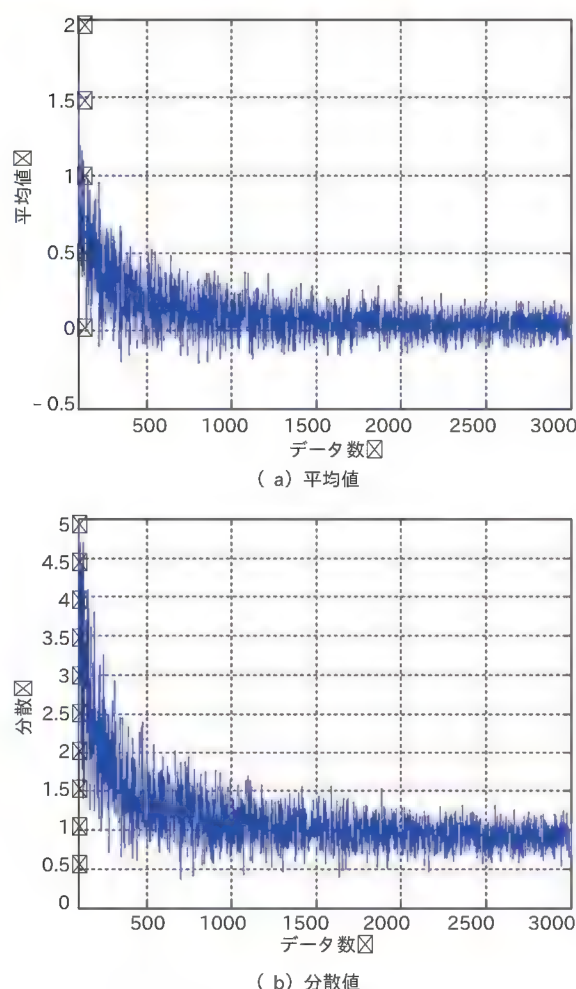
図1の1次AR過程の理論的な統計量は，平均値$E[x(n)]=0$，分散$r_{xx}(0)=E[x^2(n)]=\sigma^2$，相関関数$r_{xx}(1)=E[x(n)x(n-1)]=a\sigma^2$である[1]．$E$は期待値を表し，ここでは信号にエルゴード性を仮定して時間平均として統計量の計算を行う．□



（a）回路

（b）出力信号

図1　1次AR過程



（a）平均値

（b）分散値

図2　1次AR過程の平均値と分散

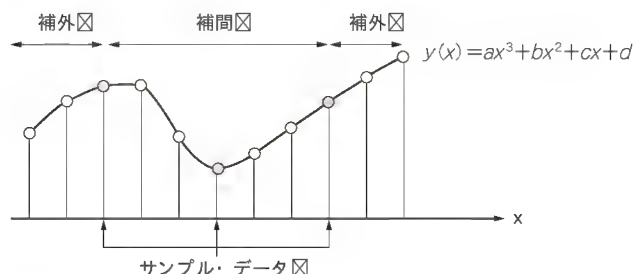


図3　補間と補外

1次AR過程に対する，理論的な相関関数の求め方は文献（1）を参照してもらうとして，図2が示す重要な点は，「少ないサンプル・データ数では，統計量が求まらない」という事実である．一致推定量が得られるサンプル数は，信号の性質によって当然ながら異なる．演習1の場合は，入力を平均と分散が規定された白色雑音としたので，出力すなわち1次AR過程も定常確率過程[注1]になっており，サンプル数が増えるにしたがって一致推定量が得られたわけである．

逆に，定常確率過程でない信号の場合は，いくらサンプル数が多くても一致推定量とならない信号もあるので注意する必要がある．一般に音声や画像などはAR過程で十分近似できることが知られている．

注1：定常確率過程とは，平均値と相関値が測定時間によらず一定値になる確率過程．

● 数学的な補間・近似方法（最小自乗，ラグランジェ，スプライン）

先の演習1において，データのサンプル数がたとえば500個しか得られず統計量が少ない場合，当然ながら一致推定量を求めることができず，図2で示すようにバイアス，すなわち誤差を含んだ平均値や相関値となる．

そこで，500個のデータからかりに1500個以上のデータを“補間”することができれば，一致推定量が得られると予測される．ここで，補間とは図3に示すように，サンプル・データ◎を使って，サンプルしていない時間におけるデータ○を推定する手法全般を指す．補間データを決定するのに使用する内挿関数 $y(x)$ を補間関数と呼び，一般に高次多項式とする場合が多い．図3の場合は3次多項式 $y(x)$ で補間した例である．

以下ではこうした目的で利用される，高次多項式を利用した信号の補間方法でよく使われ，かつ基本的な最小自乗法，ラグランジェ法，スプライン法を取り上げて，性能比較を行う．

▶ 最小自乗法（Least Square Method）

いま，補間関数 $y(x)$ を k 次多項式として，

$$y(x) = p_N x^N + p_{N-1} x^{N-1} + \cdots + p_1 x + p_0 \quad \cdots\cdots (1)$$

と定義し，さらに変数 x が x_i の時のサンプル・データを y_i として

$$J_{LS} = \sum_{i=1}^{k} \left| y_i - y(x_i) \right|^2 \quad \cdots\cdots (2)$$

なる自乗誤差評価関数 J_{LS} を $x_1 < x < x_k$ の区間で最小にする係数 p_N ～ p_0 を決定する補間方法を最小自乗法と呼ぶ．J_{LS} を最小化する係数 p_N ～ p_0 は，式(1)を各係数で微分しゼロをおけば求まる．これを正規方程式と呼び，詳細は参考文献(2)などを参照していただきたい．

最小自乗法は，基本的な補間手法だが，以下のような欠点がある．

1) サンプル・データ(離散データ)から補間関数がずれる
2) その偏移量があらかじめ規定できない
3) 式(1)を行列表示した場合に，補間点数が多いと行列サイズが大きくなり，ランク落ち(ill-condition)して正規方程式が解けない場合がある

● 演習2 最小自乗法によるデータ補間　ex2_2_lsfit.oms

以下に示す(xi, yi)，i＝1,2,…,21のデータ・セットを，3次多項式を用いて最小自乗法により補間せよ．

```
xi = -π+(i-1)π/10
yi = sin(xi)
```

解

数値演算ソフトウェア O-Matrix 注2のコマンド polyfit(x, y, n)を利用して解いた結果を図4に示す．○が与えられたデータ・サンプルyであり，実線が最小自乗法により求めた補間関数である．ここで，nは式(1)補間関数の次数である．

例題で見たように，最小自乗法の結果は，サンプル・データから補間関数がずれるという本質的な欠点がある．その意味で，最小自乗法は厳密には補間でなく，データの近似手法の一つである．

▶ ラグランジェ法(Lagrange Method)(3)

最小自乗法の欠点を解決するために，サンプル・データ(x_i, y_i)を通るように以下のように補間関数 $y(x)$ を定義する．

$$y(x) = \frac{(x-x_1)(x-x_2)\cdots(x-x_N)}{(x_0-x_1)(x_0-x_2)\cdots(x_0-x_N)} y_0 + \frac{(x-x_0)(x-x_2)\cdots(x-x_N)}{(x_1-x_0)(x_1-x_2)\cdots(x_1-x_N)} y_1 + \cdots + \frac{(x-x_0)\cdots(x-x_{k-1})(x-x_{k+1})\cdots(x-x_N)}{(x_k-x_0)\cdots(x_k-x_{k-1})(x_k-x_{k+1})\cdots(x_k-x_N)} y_k + \frac{(x-x_0)(x-x_1)\cdots(x-x_{N-1})}{(x_N-x_0)(x_N-x_1)\cdots(x_N-x_{N-1})} y_N \quad \cdots\cdots (3)$$

式(3)より，以下のことがわかる．

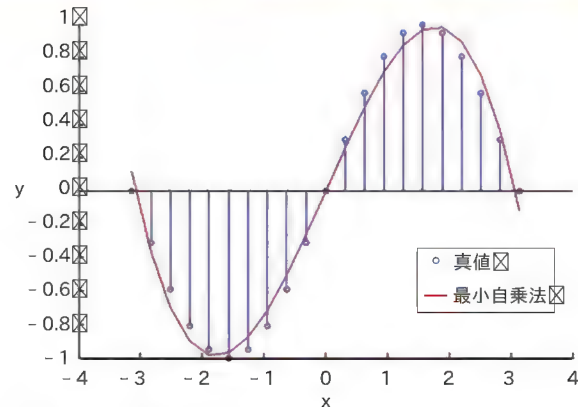


図4　演習2　最小自乗近似

1) 変数 x に x_0, x_1, …, x_N を代入すると補間関数 $y(x)$ の値は，y_0, y_1, …, y_N となりサンプル・データと一致する．これより，$y(x)$ はすべてのサンプル・データを通過する補間関数となっており，最小自乗法の欠点が解決されている
2) 分子は N 次多項式になっているので，式(3)は式(1)の補間関数と等価である

● 演習3 ラグランジェ法によるデータ補間

ex2_3_Lag_3.oms　ex2_3_Lag_13.oms

(1) 以下に示す(xi, yi)，i＝1,2,3のデータ・セットを，ラグランジェ法により補間せよ．

```
x = [-1., 0., 2.]
y = [1., 0., 4.]
```

(2) 以下に示す(xi, yi)，i=1,2,…,13のデータ・セットを，ラグランジェ法により補間せよ．

```
x = [0., 0.5, 1., 1.5, 2, 2.5, 3., 3.5, 4.,
     4.25, 4.5, 4.75, 5.]
y = [7., 5., 3., 1.5, 1., 1., 1.5, 3., 4.4,
     6., 7., 9., 10.]
```

解

数値演算ソフトウェア O-Matrix のコマンド lagrange(xd, yd, xi)を利用して解いた結果を図5に示す．○が与えられたデータ・サンプルであり，赤線が3点，緑線が14点でラグランジェ補間した結果である．□

このように，ラグランジェによる補間は，補間前のデータ・サンプル数が少ない場合は滑らかな補間関数を与えるが，サンプル数が多いと特に端点でのばらつきが大きくなるという欠点がある．ところで，ラグランジェ補間は，サンプル・データ数が少ないと‘滑らか’な補間になるので，一見少ないサンプル・データ数でも精度の高い補間が行えそうであるが，‘滑らか’さと補間・近似精度が良いこととは別なので注意すべきである．

注2：数値計算ソフト O-Matrix は，Harmonic Software Inc.より発売されている数値計算ソフトであり，行列計算，関数近似，統計処理，特殊関数などがGUIとともに豊富に提供されている．トライアル版の入手先はURLを参照のこと．http://www.omatrix.com/

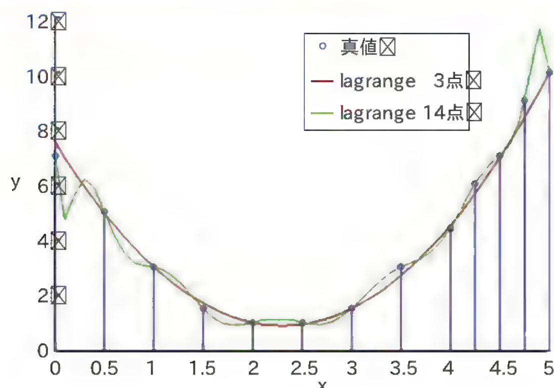


図5　演習3　ラグランジェ補間

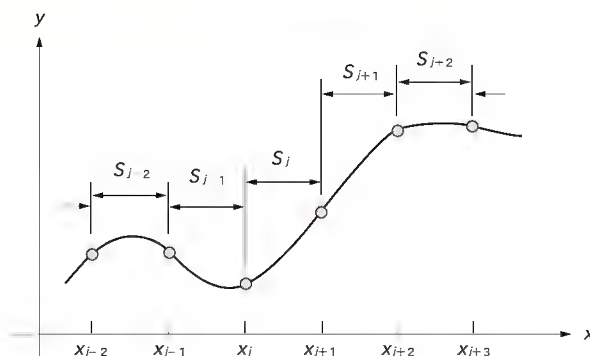


図6　スプライン補間の区分

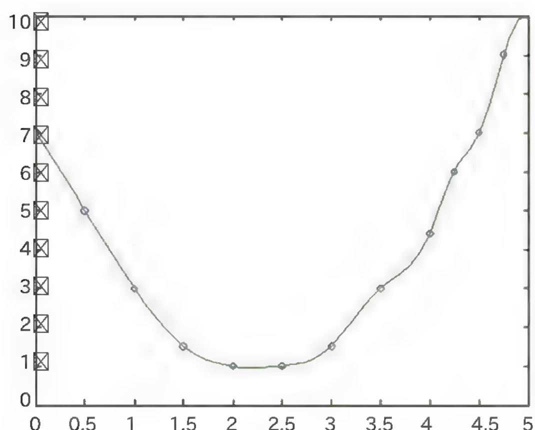


図7　演習4スプライン補間

▶ スプライン法(Spline Method)[3),(4)]

ラグランジェ法は，端点で誤差が大きくなるという欠点を有している．この欠点を解決する補間方法の一つにスプライン法がある．この方法は，図6のように近似区間xを小区間$[x_j, x_{j+1}]$にいくつか分割して，それぞれの小区間で異なった多項式関数$S_j(x)$で補間し，全区間のサンプル・データ(x_0, y_0)，(x_1, y_1)，…，(x_N, y_N)を通る滑らかな曲線を得る方法である．使用する関数は，以下の3次多項式がよく使われ，このようなスプライン補間はキュービック・スプライン(Cubic Spline)と呼ばれている．

$$S_j(x)=a_j(x-x_j)^3+b_j(x-x_j)^2+c_j(x-x_j)+d_j \quad \cdots\cdots(4)$$

スプライン法は，各小区間$[x_j, x_{j+1}]$ごとに補間関数$S_j(x)$を設定するので，境界でその導関数(微分)が不連続になる可能性がある．理由は，たとえば，$S_j(x)$と$S_{j+1}(x)$が接する境界すなわちデータ・サンプル点では，二つの曲線は接続されるが，傾き，すなわち導関数は不連続になる場合があるからである．そこで，導関数が連続になるように，各区間の係数a_j，b_j，c_j，d_jを求める必要がある．具体的には，$N-1$個の変数を有する連立方程式を解くことで求まる．

数学的な解説は，参考文献3)，4)を参照していただくとして，以下ではMATLABにより実際にスプライン補間を試してみよう．

演習4 スプライン法によるデータ補間　`ex2_4_spline.m`

以下に示す(`xi`, `yi`)，`i=1,2,…,13`のデータ・セットを，スプライン法により補間せよ．

```
x = [0., 0.5, 1., 1.5, 2, 2.5, 3., 3.5, 4.,
     4.25, 4.5, 4.75, 5.]
y = [7., 5., 3., 1.5, 1., 1., 1.5, 3., 4.4,
     6., 7., 9., 10.]
```

解

数値演算ソフトウェアMATLABのコマンド`spline`を利用して解いた結果を図7に示す．○が与えられたデータ・サンプルであり，実線がスプライン補間した結果である．□

このように，スプライン補間は，ラグランジェ法に比べて補間前のデータ・サンプル数が多くても滑らかな補間関数を与えるので，実際によく利用される．

ほかの補間・近似手法として，近似誤差の最大値を最小にするMinMax近似，誤差が等リプルとなるチェビシェフ近似などいくつか知られている[2]．

さて，これまでいくつかの高次多項式を用いた補間方法を見てきたが，どの手法がもっとも'正確'なのであろうか．どの手法も「誤差評価関数を最小」にしたり，サンプル・データ点での「近似誤差をゼロ」にしたりと，それぞれの評価尺度では最適な補間になっている．したがって，補間関数の形やその次数に依存して補間精度は左右されるので，精度の優劣に関する一般的な議論はできないというのが答である．

● マルチレート信号処理を用いた信号処理的に"正確"な補間方法

これまで補間関数をサンプル・データに'Fitting'させる方法について述べてきたが，ここでは'周波数スペクトルの近似'という側面から補間を捉えて議論を行う．この方法は，マルチレート信号処理[5),(6)]におけるインターポーレータと呼ばれる回

路(アルゴリズム)を利用する方法で，以下の特徴がある．

1) サンプル・データ，すなわちディジタル信号(離散サンプル・データ)からアナログ信号(連続信号)をサンプリング定理に基づいて復元するという意味で，最適な補間方法
2) ディジタル・フィルタを用いるのでリアルタイム性が要求される場合でも有効な補間手法

インターポレータの説明を行うにあたって必要になるフーリエ変換やマルチレート信号処理について順を追って概説する．

▶ フーリエ変換とサンプリング定理

定義1 フーリエ変換

アナログ信号 $x(t)$ をサンプル間隔 T (sec)でサンプリングしたディジタル信号 $x(nT)$ の周波数スペクトル $X(\omega)$ は次式で与えられる．ここで，n はサンプル時間を表す．

$$X(\omega)=\sum_{n=-\infty}^{\infty}x(nT)\mathrm{e}^{-j\omega nT} \quad\cdots\cdots(5)$$

ただし，$\mathrm{e}^{-j\omega nT}=\exp(-j\omega nT)$ である(指数関数；オイラーの公式)．

定理1 フーリエ変換の周期性

ディジタル信号 $x(nT)$ のフーリエ変換は，サンプリング周波数 $\omega_s=2\pi f_s=2\pi/T$ を1周期とする周期関数である．すなわち，m を整数とすると

$$X(\omega+m\omega_s)=X(\omega) \quad\cdots\cdots(6)$$

● 証明

$$X(\omega+m\omega_s)=\sum_{n=-\infty}^{\infty}x(nT)\mathrm{e}^{-j(\omega+m\omega_s)nT}$$
$$=\sum_{n=-\infty}^{\infty}x(nT)\mathrm{e}^{-j\omega nT}\mathrm{e}^{-jm\omega_s nT}$$
$$=\sum_{n=-\infty}^{\infty}x(nT)\mathrm{e}^{-j\omega nT}\ (\because \mathrm{e}^{-jm\omega_s nT}=1)$$
$$=X(\omega) \quad\cdots\cdots(7)$$

定理1を図示すると，図8となる．このように，ディジタル信号の周波数スペクトルは，f_s [Hz]ごとに繰り返すという重要な性質を有していることを覚えておいていただきたい．

定理2 アナログ信号とディジタル信号の周波数スペクトル；サンプリング定理

アナログ信号 $x(t)$ を f_s [Hz]でサンプリングしたディジタル信号を $x_d(nT)$ とする．$x_d(nT)$ のスペクトル $X_d(\omega)$ は，$x(t)$ の周波数スペクトル $X(\omega)$ が f_s [Hz]ごとに繰り返す周期関数となる．

$$X_d(\omega)=\frac{1}{T}\sum_{k=-\infty}^{\infty}X(\omega+k\omega_s) \quad\cdots\cdots(8)$$

□

定理2の証明は省略するが[5]，定理1の図8においてゼロ周波数付近のスペクトルをアナログ信号 $X(\omega)$ と考えると，直感的に定理2は理解できるであろう．この定理は，サンプリング(標本化)定理と呼ばれ，ディジタル信号処理の中核をなす重要な定理である．

サンプリング周波数 f_s [Hz]ごとに現れるスペクトルは，イメージング(imaging)成分と呼ばれる．

この定理より，次のような補間方法が直ちに考えられる．すなわち，ディジタル信号の周期性を持つ周波数スペクトル $X_d(\omega)$ からゼロ周波数付近のベース・バンド・スペクトル $X(\omega)$ のみ取り出すことができるならば，ディジタル信号すなわちサンプル・データ $x_d(nT)$ からアナログ信号 $x(t)$ を再生することができ，補間が可能であることがわかる．

この補間方法について検討するため，まずサンプル・レートを下げる方法から考えていく．

▶ ダウン・サンプラとデシメータ

例題1 サンプル・レートを低くする

図9(a)に，あるアナログ信号の時間波形 $x(t)$ と周波数スペクトル $X(\omega)$ を示す．$x(t)$ をサンプル・レート(サンプリング周波数) $f_s=4$ [kHz]および $f_s'=f_s/M$ によりサンプリングした場合の周波数スペクトルを描いてみよう．ただし，$M=2$ とする．

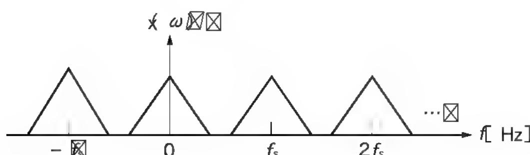
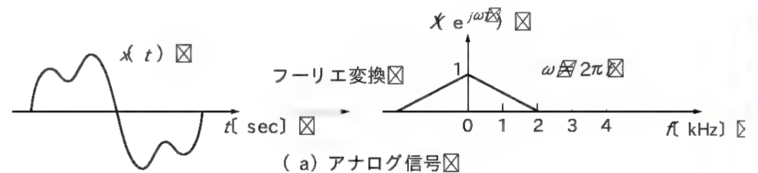
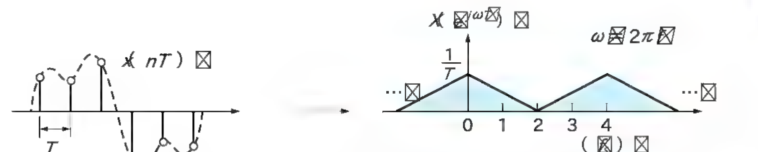


図8 フーリエ変換の周期性





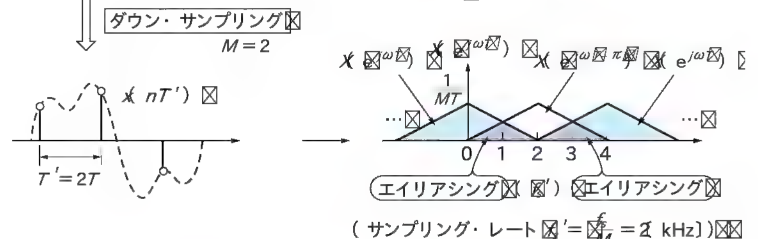


図9 ダウン・サンプルと周波数スペクトル

解

結果を図9(b)，(c)にそれぞれ示す．この結果は，定理2のサンプリング定理より容易に理解できるであろう．ただし，振幅値がそれぞれ $1/T$，$1/MT$ となる．$T = 1/f_s$ である．

$x(nT)$ と $x(nT')$ は，異なるサンプル・レートによりサンプリングして得られた信号だが，ここでは，"$x(nT)$ のサンプルを $M-1$ 個間引いて，M サンプルごとの信号のみを取り出して $x(nT')$ を得る"と考えてみよう．この操作を，図10のダウン・サンプラ(Down-sampler)と呼ばれるブロックで表すことにする．Mをダウン・サンプリング率と呼ぶ．

ところで，図9(c)においてオーバラップしている部分は，ダウン・サンプルにより生じたスペクトルでありエイリアシング(aliasing)成分と呼ばれる．エイリアシングが生じると元のアナログ信号のスペクトル情報が失われるので好ましくない．

以下では，エイリアシングをしないようにサンプル・レートを低減する方法について検討する．

まず，ダウン・サンプルで生じるエイリアシングを消去するためには，図9(b)の周波数スペクトルにおいて，帯域制限を行えばよい．このため，図11(a)のデシメータ(Decimator)と呼ばれる回路ブロックを使用する．デシメータで使われるディジタル・フィルタ(Low Pass Filter：LPF)をとくにデシメーション・フィルタと呼ぶ．このようにデシメータによりサンプル・レートが低減される．有限長インパルス応答タイプ(FIR)のディジタル・フィルタは，MATLABでは`fir1`などのコマンドで設計できる．

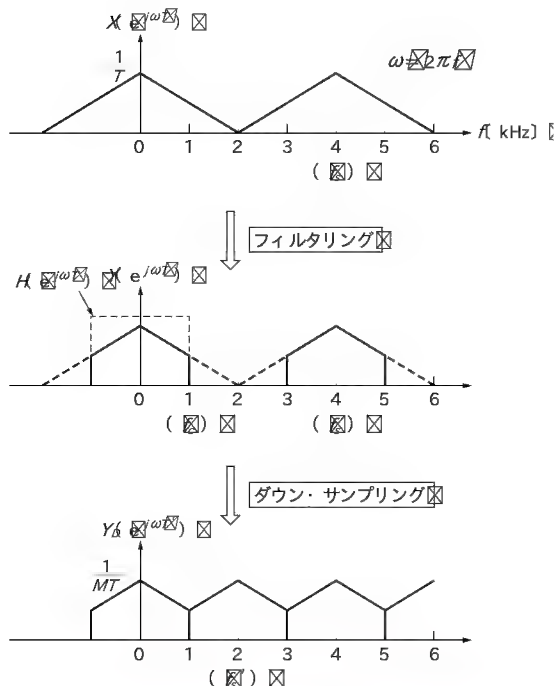


図12　デシメーションと周波数スペクトル

例題2 デシメーションと周波数スペクトル

例題1において，エイリアシングが生じないようにデシメータを用いてサンプル・レートを1/2にしたい．デシメーション・フィルタのカット・オフ周波数 f_c を決定し，図11の各部の周波数スペクトルを描いてみよう．

解

$M = 2$ なので，$f_c = f_s/2M = 1$[kHz]と設定する．各部の周波数スペクトルを図12に示す．

ところで，デシメータで得られる信号 $y_D(nT')$ は，デシメーション・フィルタによる帯域制限により元の信号 $x(nT)$ の情報を失っている点に注意しなければならない．

▶ アップ・サンプラとインターポレータ

つぎにサンプル・レートを上げる方法について考える．

例題3 サンプル・レートを上げる

図13(a)の信号 $x(nT)$ のサンプル間に $L-1$ 個の零値を挿入

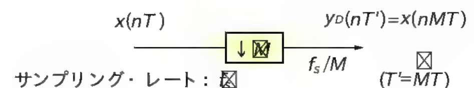
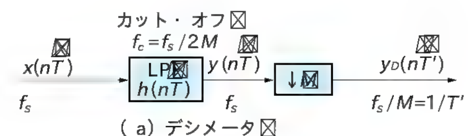
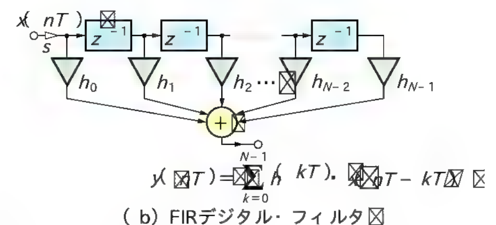


図10　ダウン・サンプラ

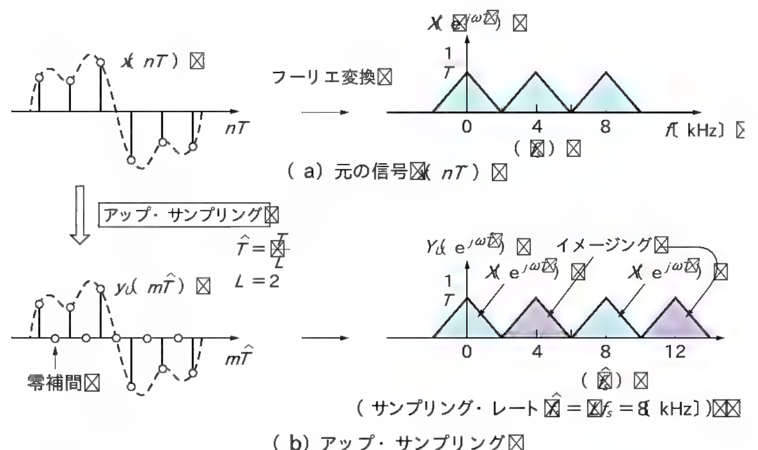


(a) デシメータ



(b) FIRデジタル・フィルタ

図11　デシメータ



(b) アップ・サンプリング

図13　アップ・サンプルとインターポレーション

し，図13(b)の$y_u(m\hat{T})$を得る．サンプル間隔$\hat{T}=T/L$なので，$y_u(m\hat{T})$は$x(nT)$に比べてサンプル・レートがL倍に上がっている．$y_u(m\hat{T})$の周波数スペクトルを描いてみよう．

解

$y_u(n\hat{T})$のフーリエ変換を求める．式(5)より

$$
\begin{aligned}
Y_u(\omega) &= \sum_{m=-\infty}^{\infty} y_u(m\hat{T})e^{-j\omega m\hat{T}} \\
&= \sum_{n=-\infty}^{\infty} y_u(nL\hat{T})e^{-j\omega nL\hat{T}} \quad (\because m = nL) \\
&= \sum_{n=-\infty}^{\infty} x(nT)e^{-j\omega nT} \quad (\because T = L\hat{T}) \\
&= X(\omega) \quad \cdots\cdots (9)
\end{aligned}
$$

となり，図13(b)に示すように入力信号$x(nT)$のスペクトル$X(\omega)$と変わらないことがわかる．

アップ・サンプルしてサンプル・レートがL倍になると，本来はサンプリング周波数$\hat{f}_s=1/\hat{T}$に最初の折り返しスペクトルが生じると錯覚しがちだが，例題で示したようにアップ・サンプリングする前のスペクトルと変わらない．これは，ゼロを補間しても情報を有していないので，何の変化も起きないのである．

以上のように，$L-1$個の零値を補間するブロックをアップ・サンプラ(Up-sampler)と呼び，図14のブロックで表す．Lをアップ・サンプリング率と呼ぶ．

次に，アップ・サンプルで生じるイメージングを消去するためには，図13(b)の周波数スペクトルを帯域制限すればよい．このため，図15(a)のインターポーレータ(Interpolator)と呼ばれる回路ブロックを使用する．このディジタル・フィルタ(LPF)をとくにインターポーレーション・フィルタと呼ぶ．

インターポーレータのディジタル・フィルタはsinc関数のインパルス応答を有するので，インターポーレータは，補間関数をsinc関数とする補間方法と考えられる．

例題4 インターポーレーションと周波数スペクトル

例題3において，イメージングが生じないようにインターポーレータを用いてサンプル・レートを2倍に上げたい．インターポーレーション・フィルタのカット・オフ周波数f_cを決定し，図15(a)の各部の周波数スペクトルを描いてみよう．

解

$$f_c = f_s/2 = 2\ \mathrm{[kHz]}$$

周波数スペクトルを図15(b)～(d)に示す．

本例題より，以下の重要なことがわかる．インターポーレータでイメージング成分を消去することにより，アナログ信号をサンプル・レート$\hat{f}_s=1/\hat{T}$でサンプリングした場合と同様の周波数スペクトルが得られる．つまり，時間領域では図15(d)で示すように，零補間したサンプル点に$y_i(m\hat{T})$が現れる(補間される)のである．このことが，インターポーレータの名前の由来である．

インターポーレータは，MATLABにおいてコマンドinterpあるいはresampleとして提供されている．

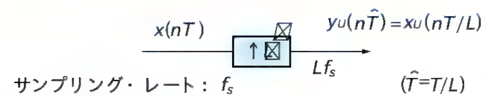
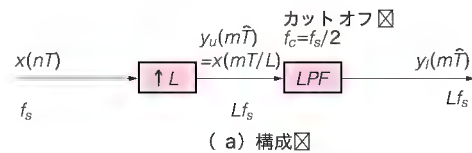
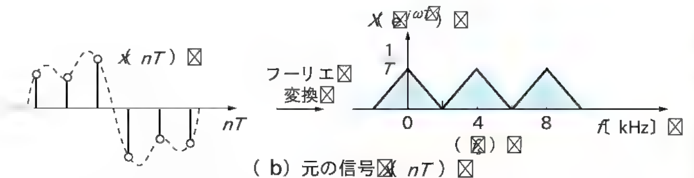
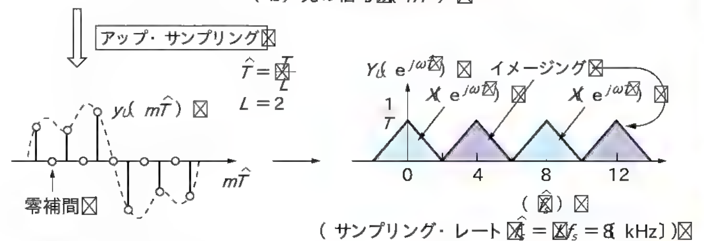


図14 アップ・サンプラ







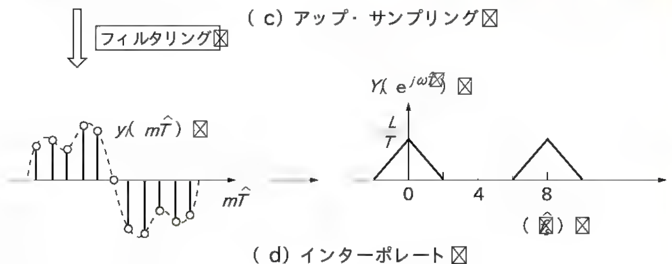


図15 インターポーレータと周波数スペクトル

演習5 1次AR過程の統計量 ex2_1_5.m

演習1で扱った1次AR過程において，インターポーレータを用いて少ないデータ数で相関係数$r_{xx}(1)$を推定してみよう．アップ・サンプリング率$L=2, 5, 10$とする．

解

図16に結果を示す．理論的には$r_{xx}(1)=a=0.9$であるので，1500サンプル程度で一致推定量が得られていることがわかる．この結果は，演習1で検討した平均値・分散と同様である．

ここで，1500サンプルの時点において，$L=2$の場合は750，$L=5$では300，$L=10$では150サンプルだけ用いて補間し，1500サンプル分を求めていることに注意してほしい．よって，インターポーレータを用いると，少ないデータ数でも相関係数が求まることがわかる．ただし，$L=10$とアップ・サンプリング率が上がると補間精度は低下する．

演習6 ラグランジェ補間とインターポーレータを用いた補間の精度比較――日経平均終値 ex2_6_stock.m

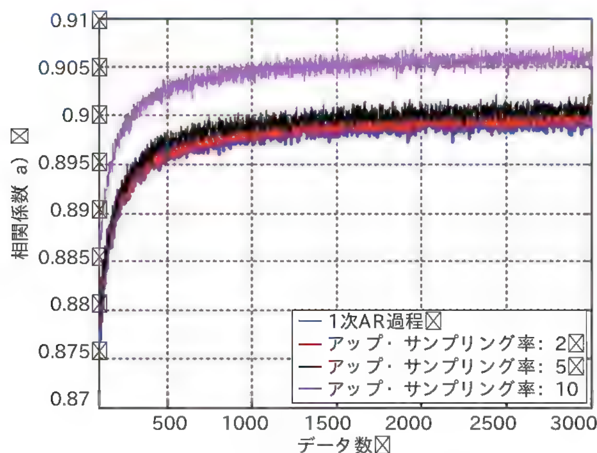


図16　インターポレーションと1次AR過程相関関数

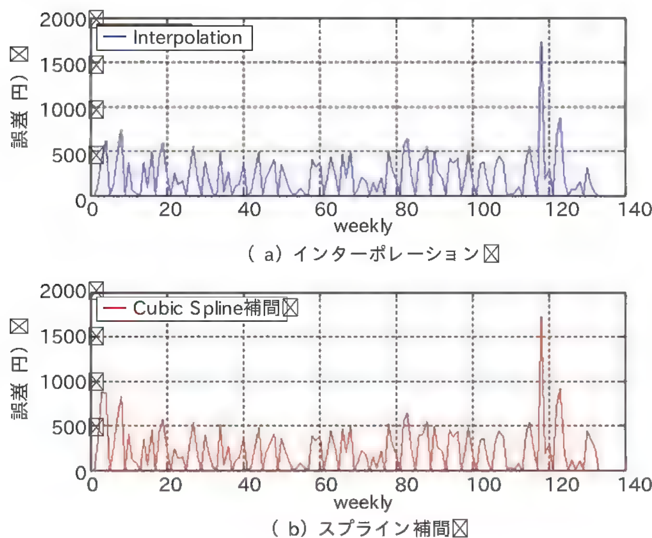


図18　日経平均終値の補間誤差

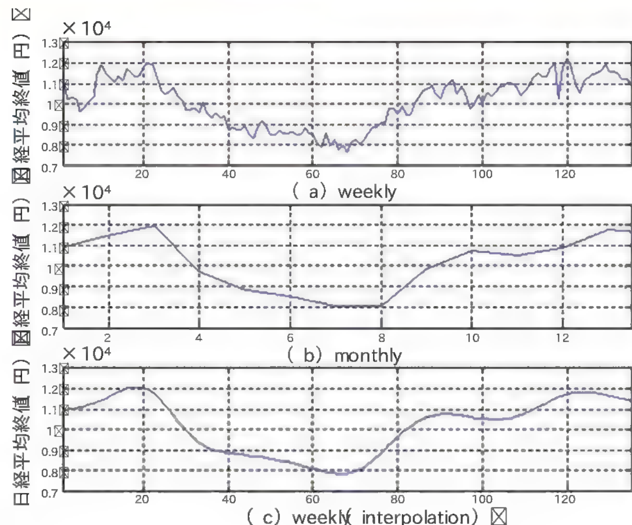


図17　日経平均終値

図17(a)に2002年1月7日(月)～2004年8月9日(月)の135週分の日経平均終値のデータを示す．ここで，1か月ごと(4週おき)にしかデータがない場合を想定し[図17(b)]，インターポレーションおよびスプライン補間を用いて週データの補間を行え．

解

図17(c)にアップ・サンプリング率4のインターポレーションによって補間したデータを示す．図18は，元々の日経平均終値とインターポレーション，スプライン補間との絶対誤差をそれぞれ示している．このように，インターポレータを用いた補間とスプライン補間は，ほぼ同じ程度の補間精度であり，最大1,700円程度の補間誤差となっていることがわかる．

なお，インターポレータの補間精度は，LPFの設計に依存する．すなわち，イメージング成分を十分に抑圧できれば(＝フィルタの周波数特性において阻止域減衰量を十分取れば)精度は上がるが，そうでないと精度が悪くなる．また，サンプル・データが帯域制限されていないとエイリアシングが生じている可能性が高く，インターポレータで補間が十分できない場合もあるので注意すべきである．

少ないデータ数でスペクトル解析をする

定義1のフーリエ変換は，時間信号$x(nT)$は離散的であるが，周波数は連続量ωであり，計算機向きではない．そこで，離散周波数で定義して計算機で計算しやすくしたフーリエ変換，すなわち離散フーリエ変換(DFT)が一般によく使われる[5]．DFTの高速解法が，有名な高速フーリエ変換(FFT)である．

本項では，このDFTを用いた周波数スペクトル解析において，少ないサンプル・データでも周波数分解能を高くするいくつかの手法について解説する．

離散フーリエ変換$X(k)$は，サンプル・データ$x(n)$，$n=0, 1, \cdots, N-1$に対して次式で与えられる．

定義2　離散フーリエ変換DFT

$$X(k)=\sum_{n=0}^{N-1}x(n)e^{-j\frac{2\pi nk}{N}} \quad \cdots\cdots (10)$$

ただし，$k=0, 1, \cdots, N-1$

周波数スペクトル$X(k)$から時間信号(サンプル・データ)$x(n)$を求める逆変換は，

定義3　逆離散フーリエ変換IDFT

$$x(n)=\frac{1}{N}\sum_{k=0}^{N-1}X(k)e^{j\frac{2\pi nk}{N}} \quad \cdots\cdots (11)$$

● ゼロ・パッド法

DFTは信号$x(n)$のサンプル数Nが決まると，周波数分解能は定義2より$2\pi/N$[rad・Hz]$=f_s/N$[Hz]，f_s：サンプリン

グ周波数)と自動的に決まってしまう．つまり，分解能を上げたくても，$2\pi/N$より上げられない．さて，この周波数分解能を上げる方法がないものだろうか．

そこで，Nサンプルの$x(n)$の後ろにゼロを$M-N$個追加して，Mサンプルの信号$x'(n)$を作ることを考える．すると，$x'(n)$のDFT　$X'(k)$は，

$$X'(k)=\sum_{n=0}^{M-1}x'(n)e^{-j\frac{2\pi nk}{M}}$$
$$=\sum_{n=0}^{N-1}x(n)e^{-j\frac{2\pi nk}{M}} \quad \cdots\cdots(12)$$

$$\because x'(n)=\begin{cases}x(n), & 0\leqq n\leqq N-1\\ 0, & N\leqq n\leqq M-1\end{cases}$$

$X'(k)$の周波数分解能は$2\pi/M$となり，NポイントDFTの分解能$2\pi/N$より上がっていることがわかる(図19)．

このように，ゼロ・パッド(追加)するだけで周波数分解能が上がるので，データ数Nを増やすことができない場合に有効である．この方法をゼロ・パッド法(Zero-Padded Method)と呼んでいる．

演習7 DFTの周波数分解能を上げる

ex3_1_zeropadDFT.m

次の正弦波について，DFTによりスペクトル解析を行ってみたい．(a)，(b)それぞれの条件で実行してみよう．なお，窓関数は使用しない．

$x(n)=\sin(2\pi fn)$，　$0\leqq n\leqq N-1$

ただし，$f=0.1$[Hz]，$N=16$とする．

(a) 16点DFT

(b) 64点DFT．ただし，$N\leqq n<64$で$x(n)=0$とゼロ・パッドする．

解

結果を図20に示す．このように周波数分解能を上げること

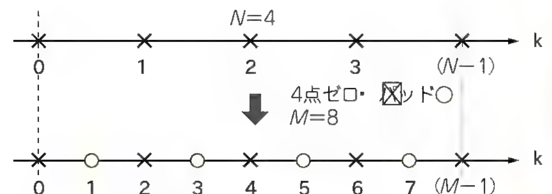
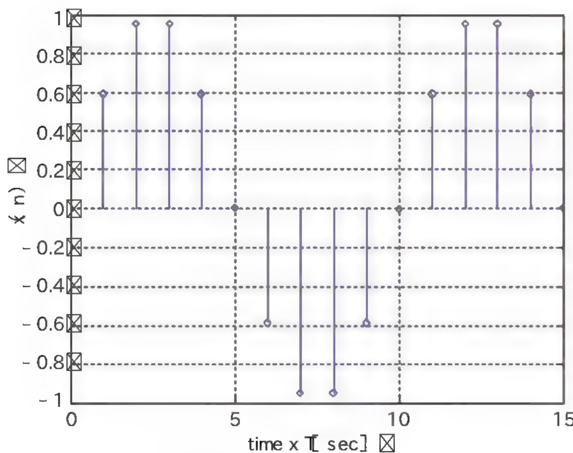
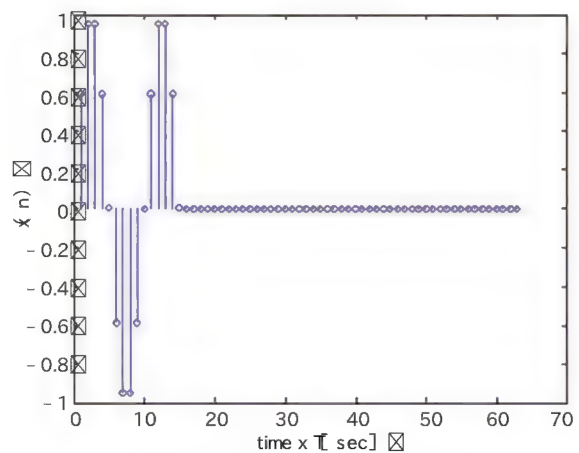


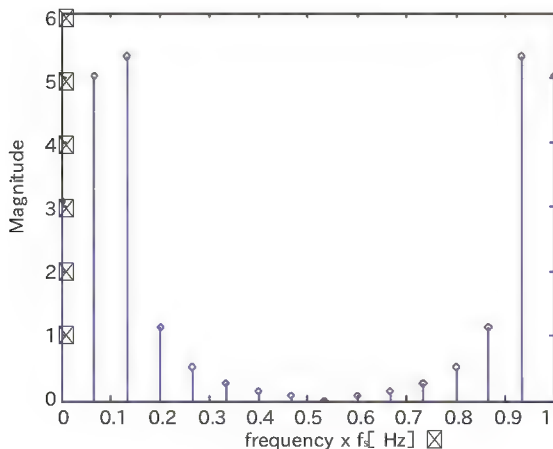
図19　ゼロ・パッド法の周波数点



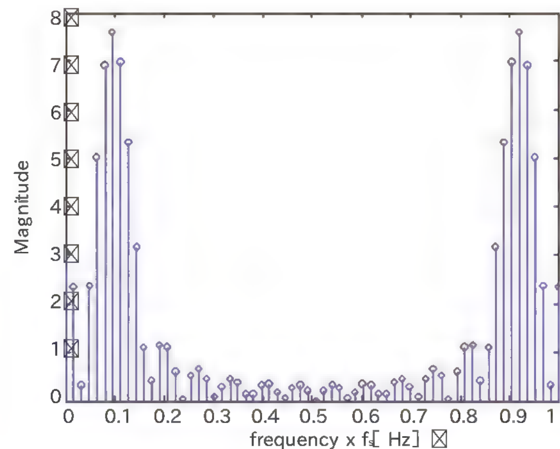
(a) 16点サンプル・データ



(b) ゼロ・パッドした64点サンプル・データ



(c) 16点DFTスペクトル



(d) 64点DFTスペクトル

図20　演習7　ゼロ・パッド法によるスペクトル解析

により，少ないデータ数の 16点では見えなかった 0.1Hz 付近のスペクトルが見えてくるようになることがわかる．

● ズーミング変換

つぎに，ズーミング変換(Zooming Transform)と呼ばれるマルチレート信号処理を用いた手法を紹介する．この手法は，特定の周波数区間の周波数分解能を上げることができる．

図 21(a)にズーミング変換の一回路を示す．入力は，N 点 DFT 周波数スペクトル $X_1(k)$であり，この時点での分解能は f_s/Nである．まず，$X_1(k)$が IDFT により時間信号 $x(n)$に戻され，$f(n)$により $-f_c$だけ周波数シフトされる．ついで，LPF の $h(n)$により図 21(b)のように帯域制限される．ここで，$h(n)$は複素数のインパルス応答である．

LPF 後はエイリアシングが生じないように M の値を調整してダウン・サンプルする．最後に，P 点 DFT を施すと，周波数区間 $f_c - \Delta f < f_c < f_c + \Delta f$のスペクトルが分解能 $2\Delta f/P$ で求まる．このように特定の周波数区間のスペクトルをズーミングして，かつ分解能をあげることが可能である．

● 不等間隔 DFT

ところで，式(10)で定義された DFT は周波数間隔が等間隔であり，たとえばゼロ・パッド法を用いていくら分解能を上げても，測定したい周波数に一致しない場合がある．

そこで，以下では任意の周波数点でスペクトル解析が可能な，不等間隔 DFT(Nonuniform DFT)について解説する(7)．

▶NDFT と逆変換

定義4　不等間隔 DFT(NDFT)

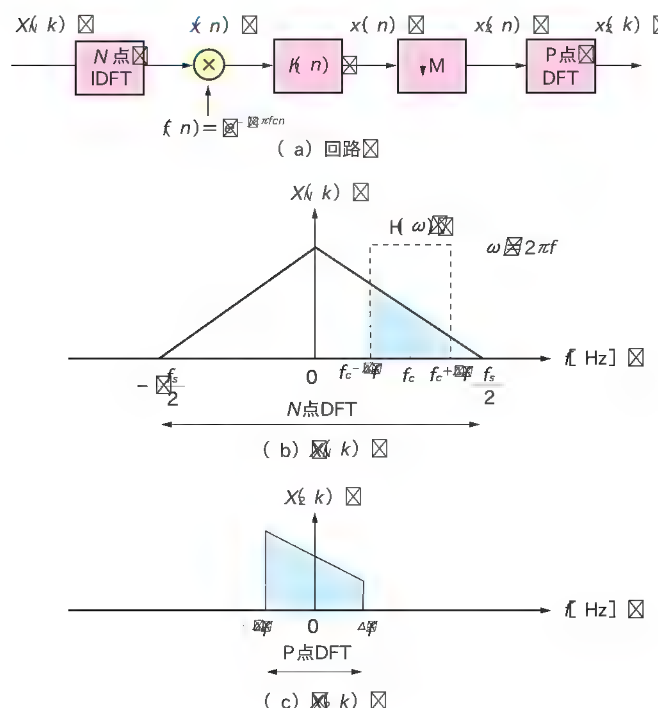


図 21　ズーミング変換

$$X(z)\Big|_{z=z_k} = \sum_{n=0}^{N-1} x(n) z_k^{-n},\ k=0,1,\cdots,N-1 \quad \cdots\cdots(13)$$

ここで，$X(z)$は $x(n)$の z 変換であり，z_0, z_1, …, z_{N-1}は z 平面上の任意の異なった点である．$z_k = e^{-j\omega k}$と単位円に取った場合を不等間隔 DFT と呼んでいる．

任意の周波数点で解析できるフーリエ変換として上記のように NDFT を定義したが，スペクトルから時間信号に戻すためには逆変換が定義できなければならない．この問題に対して，まず式(13)を行列で書き換えると，

$$\boldsymbol{X} = \boldsymbol{D}\boldsymbol{x} \quad \cdots\cdots(14)$$

ただし，

$$\boldsymbol{X} = \begin{bmatrix} X(z_0) \\ X(z_1) \\ \vdots \\ X(z_{N-1}) \end{bmatrix},\ \boldsymbol{x} = \begin{bmatrix} X(0) \\ X(1) \\ \vdots \\ X(N-1) \end{bmatrix}$$

$$\boldsymbol{D} = \begin{bmatrix} 1 & z_0^{-1} & z_0^{-2} & \cdots & z_0^{-(N-1)} \\ 1 & z_1^{-1} & z_1^{-2} & \cdots & z_1^{-(N-1)} \\ \vdots & & & & \vdots \\ 1 & z_{N-1}^{-1} & z_{N-1}^{-2} & \cdots & z_{N-1}^{-(N-1)} \end{bmatrix}$$

となる．行列 $\boldsymbol{D}$ は Vandermonde 行列と呼ばれ，以下のように z_0, z_1, …, z_{N-1}が z 平面上で異なった点であるならば，以下の行列式が存在する．よって，逆行列が存在するので定義 5の NDFT の逆変換が定義できる．

$$\det(\boldsymbol{D}) = \prod_{i\neq 1,\, i>j} \left(z_i^{-1} - z_j^{-1}\right) \quad \cdots\cdots(15)$$

定義5　逆 NDFT

$$\boldsymbol{x}_1 = \boldsymbol{D}^{-1}\boldsymbol{X} \quad \cdots\cdots(16)$$

任意の周波数のスペクトルが解析でき，かつ逆変換も存在する NDFT の性質は，数学的にも興味深い．

演習8　特定の周波数を解析する

—— Nonuniform DFT　ex3_2_NDFT.m

以下の条件の正弦波信号に対して，

(1) DFT と NDFT の周波数スペクトルの比較(窓関数は用いない)

(2) 逆 NDFT を用いて，周波数スペクトルから時間信号に戻るかを確認せよ

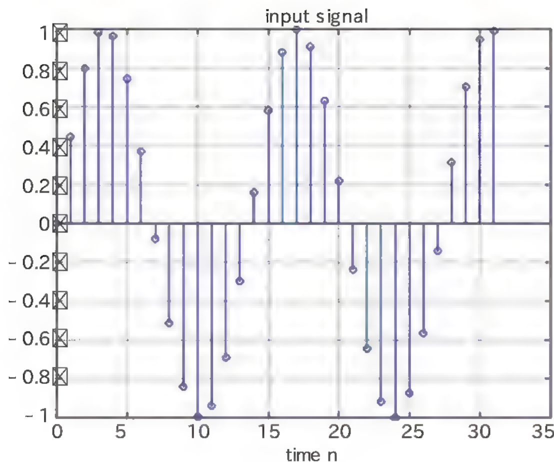
DFT 点数 N = 32

サンプリング周波数 f_s = 4096Hz

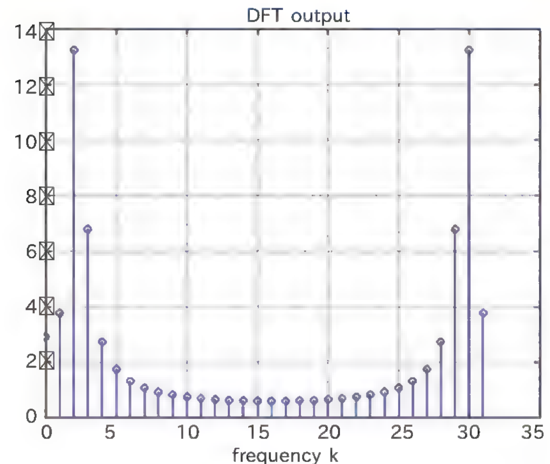
発振周波数 f = 300Hz

解

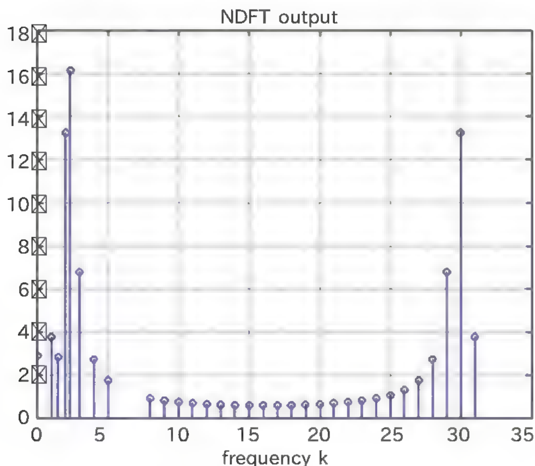
図 22に結果を示す．NDFT は，DFT に比べて低域に周波数解析点を不等間隔で増やしている．このように図 22(b)の DFT では，300Hz に相当する周波数点が存在しないので正弦波のサイド・ローブしか解析できていない．一方，NDFT の場合は正確に 300Hz を指定できるので，図 22(c)のように振幅値が 16($N/2$= 16がピーク値となる)となり，正弦波そのものが解析



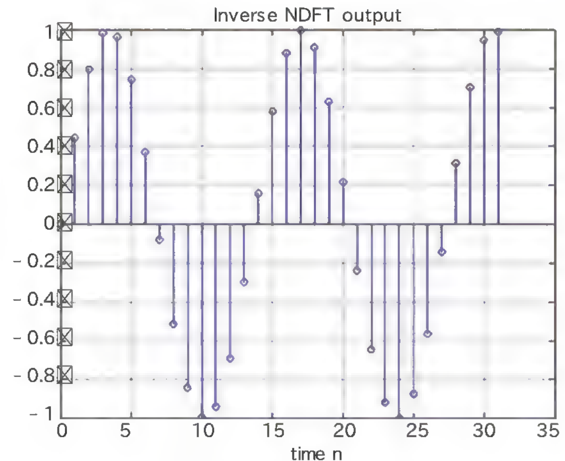
（a）原信号 $f(n)=\sin(2\pi fnT)$，$T=1/fs$



（b）DFT 出力（振幅スペクトル）



（c）NDFT 出力（振幅スペクトル）



（d）Inverse NDFT による原信号の再生

図 22　演習 8　不等間隔 DFT（NDFT）

できていることがわかる．

また，図 22（d）と図 22（a）が一致しているので，逆変換が式（15）で求まることがわかる．

ところで，NDFT は不等間隔周波数なので，高速演算アルゴリズムである FFT は利用できないが，IIR ディジタル・フィルタを用いた Goertzel アルゴリズムにより少ない演算量で求めることができる(7)．



雑音の影響を少なくする

さて，話題を変えて，信号の不連続性検出と雑音除去に関して議論することにする．解析対象のサンプル・データ信号に，突発的に，たとえば直流が付加されたり周波数が変化するなどの不連続性が生じたり，あるいは雑音が付加されている状態では，統計量推定やスペクトル推定は正確に行えない．以下では，こうした問題を解決するウェーブレット（Wavelet）変換を用いた信号の不連続性検出と雑音除去の方法について解説していく．

ウェーブレット変換は数学的に捉えるとかなり難解だが(8),(9)，信号処理論的に解釈するとディジタル・フィルタとして表現でき，物理的な理解がしやすく応用上も重要である(6),(10)．逆にいうと，ウェーブレット変換は厳密な理論はわからなくても，「ディジタル・フィルタで実現できる直交変換である」という理解だけで十分である．本章やシミュレーション・プログラムで‘ウェーブレットが使える’ようになるはずである．

● 短時間フーリエ変換

時間的に周波数が変化したり，値が不連続になる信号 $x(t)$ をフーリエ変換して周波数スペクトル $X(\omega)$ を求めても，どの時間で周波数が変化したか，信号値が不連続になったか検出できない．理由は，$X(\omega)$ には時間変数が失われているからである．

▶ 短時間フーリエ変換

そこで，短時間フーリエ変換（Short time Fourier transform）

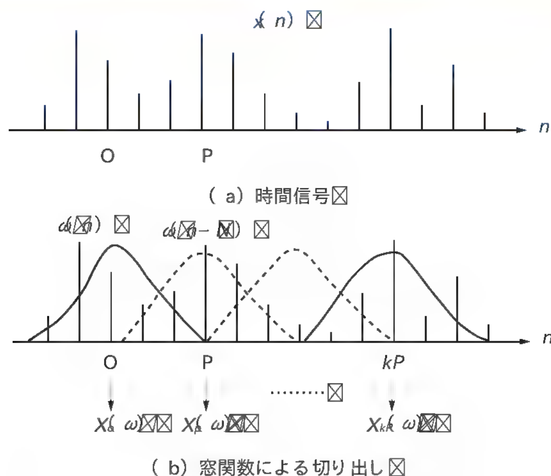


図23　窓関数 $w(n)$ のシフトによる信号 $x(n)$ の切り出し

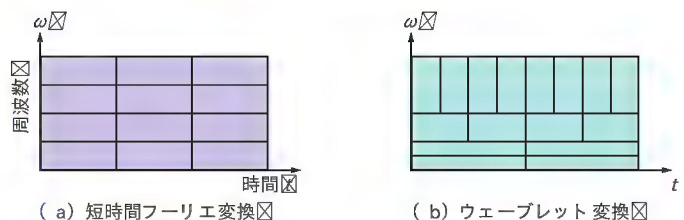


図24　時間・周波数分解能（タイリング）

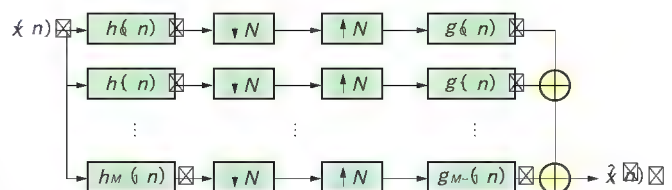


図25　短時間フーリエ変換のフィルタ・バンク表現

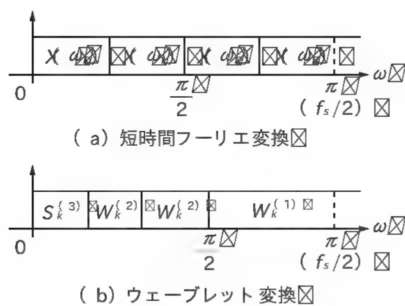


図26　フィルタ・バンクの周波数特性（$M=4$）

と呼ばれる，時間的に窓関数をシフトして時間変数を残した時間－周波数解析が可能なフーリエ変換を考えてみよう．

いま，図23のように窓関数 $w(n)$ を P サンプルずつシフトしながら，信号 $x(n)$ を時間的に順次切り出すことにする．k ブロック目の信号は $x(n)w(n-kP)$ と表されるので，そのフーリエ変換は式(5)より，

$$X_{kP}(\omega)=\sum_{n=-\infty}^{\infty}x(n)w(n-kP)\mathrm{e}^{-j\omega n} \quad \cdots\cdots\cdots\cdots(17)$$

と時間変数(kP)を含むフーリエ変換になる．これを短時間フーリエ変換と呼んでいる．$T=1$と規格化している．

フーリエ変換が $\mathrm{e}^{-j\omega n}$ を基底とする直交変換であることに対し，短時間フーリエ変換は，$w(n-kP)\mathrm{e}^{-j\omega n}$ を基底とする直交変換になっているとも考えられる．

▶ 短時間フーリエ変換とウェーブレット変換の分解能

時間－周波数解析から見ると，短時間フーリエ変換はどの周波数でも同じ時間長の基底を使っている（図23）ことに対し，ウェーブレット変換は周波数によって異なった時間長の基底を使用する．したがって，時間－周波数分解能を考えると，図24で示すように短時間フーリエ変換はどの時間－周波数でも一定であるのに対し，ウェーブレット変換は任意の時間・周波数で自由に分解能を設定できる．低周波数域の分解能を順次高く設定する場合をウェーブレット変換と呼び，信号の性質に応じてまったく自由に設定する場合をウェーブレット・パケット（Wavelet Packet）と呼ぶ．詳しくは後述する．

▶ フィルタ・バンク表現

M 点の周波数（ω_i，$i=0, 1, \cdots, M$）で短時間フーリエ変換を求めるとすると，式(17)は以下のように変形できる．

$$\begin{aligned}X_{kP}(\omega_i)&=\mathrm{e}^{-j\omega_i kP}\sum_{n=-\infty}^{\infty}x(n)w(n-kP)\mathrm{e}^{j\omega_i(kP-n)}\\&=\mathrm{e}^{-j\omega_i kP}\sum_{n=-\infty}^{\infty}x(n)h_i(kP-n)\\&=\mathrm{e}^{-j\omega_i kP}\{x(kP)*h_i(kP)\} \quad \cdots\cdots\cdots(18)\end{aligned}$$

ただし，$h_i(kP)=w(-kP)\mathrm{e}^{j\omega_i kP}$

最後の＊印は畳み込み演算を表しており，インパルス応答が $h_i(kP)$ で入力信号が $x(kP)$ のディジタル・フィルタで実現できることを示している．このように，M 点の離散周波数 ω_i を計算する短時間フーリエ変換は，M 分割分析フィルタ・バンク（アナライザ）と等価になることがわかる．

周波数 ω_i が異なってもインパルス応答 $h_i(kP)$ の長さは変わらないので，各フィルタ $X_{kP}(\omega_i)$ の周波数帯域幅は同じである．仮に ω_i を等間隔にとり $M=4$とすると，各フィルタの周波数帯域は，図26(a)となる．

一方，スペクトル $X_{kP}(\omega_i)$ から時間信号 $x(n)$ に逆変換する合成フィルタ・バンク（シンセサイザ）は，アナライザと同様に導出でき，アナライザとシンセサイザを結合した回路が図25である．式(18)の $\mathrm{e}^{-j\omega_i kP}$ はシンセサイザで生じる $\mathrm{e}^{j\omega_i kP}$ で相殺される[10]．短時間逆フーリエ変換により時間信号 $x(n)$ が再生できるかどうかという問題は，フィルタ・バンクの完全再構成問題となり，したがってインパルス応答 $h(n)$，$g(n)$ の設計問題に帰着する．

● ウェーブレット変換

さて，これまで説明してきた短時間フーリエ変換と対比させて，ウェーブレット変換を説明する．ここでは離散ウェーブレット変換をウェーブレット変換と呼ぶことにしておく．

▶ 時間−周波数分解能

先に図24で説明したように，時間−周波数分解能を考えると，図24（a）で示すように短時間フーリエ変換はどの時間−周波数でも一定であるのに対し，ウェーブレット変換は任意の時間−周波数分解能を設定できる．低周波数域の分解能を順次高く設定する場合がウェーブレット変換であり，4チャネルの場合を図24（b）に示す．いいかえれば，フィルタのインパルス応答の長さを解析周波数によって変えていることとなり，その概念図を図27に示す．このように，ウェーブレット変換は，時間と周波数の両方に局在することができ，短時間フーリエ変換と異なる性質を有する．

低周波数域の分解能を順次上げていく理由は，音声や音響などのエネルギが低周波に集中する場合が多いからである．したがって，ウェーブレット変換が圧縮符号化や時間周波数解析に適していることが理解できる．

▶ フィルタ・バンク表現

それでは，図24（b）のような周波数分解能を持つアナライザ・フィルタ・バンクについて検討してみよう．まず，図28に示す低域通過フィルタ $P(z)$ と高域通過フィルタ $Q(z)$ を並列させた2分割フィルタ・バンクを考える．ここで，$P(z)$，$Q(z)$ は z 変換の伝達関数を示し，$z = e^{j\omega T}$ とおくとフーリエ変換，すなわち周波数特性となる[5]．

この2分割フィルタ・バンクをバイナリ・ツリー状に低域側をさらに分解していく．たとえば，3レベルまで分解すると図29（a）のように4チャネルのフィルタ・バンクが構成でき，その周波数特性は図26（b）となることが容易に理解できるであろう．各フィルタの出力でダウン・サンプルできるのは，図28（b）で示すように帯域制限されているからである．

逆ウェーブレット変換すなわちシンセサイザ・フィルタ・バンクは，図29（b）となる．

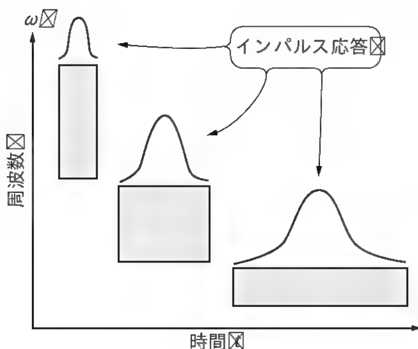


図27　ウェーブレット変換の時間・周波数局在性

図29を並列表現すると，図30となる[10]．図30のウェーブレット変換における各フィルタの伝達関数 $A_i(z)$ はそれぞれ異なるので，インパルス応答も帯域 i によってその長さが異なる．この点が，図25の短時間フーリエ変換とウェーブレット変換が異なる点である．したがって，時間・周波数分解能が図24（b）となることが理解できる．

▶ 2分割完全再構成フィルタ・バンク − CQF −

さて図29のフィルタ・バンクがウェーブレット変換・逆変換と等価になる条件の一つとして，遅延を許して入力信号 $f(n)$ が出力で完全に再生されなければならない（完全再構成条件）．そのためには，図29のフィルタ・バンクの基本構成要素である図28の2分割フィルタ・バンクが完全再構成である必要がある．以下に，その設計条件を簡単に述べる．

図28において入力と出力の z 領域における関係は以下のようになる[5]．

$$\hat{F}(z) = \frac{1}{2}\{P(z)F(z) + P(-z)F(-z)\}R(z) + \frac{1}{2}\{Q(z)F(z) + Q(-z)F(-z)\}X(z) \quad \cdots\cdots(19)$$

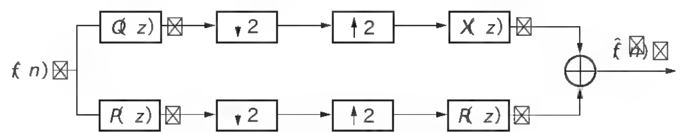
$$= \frac{1}{2}\{P(z)R(z) + Q(z)X(z)\}F(z) + \frac{1}{2}\{P(-z)R(z) + Q(-z)X(z)\}F(-z) \quad \cdots\cdots(20)$$

上式の2行目の右辺第2項はダウン・サンプリング/アップ・サンプリングによって発生したエイリアシング，およびイメージング成分である．ここで，信号を再構成させるためには次式が成立する必要がある．この場合，L サンプルの遅延を許容している．

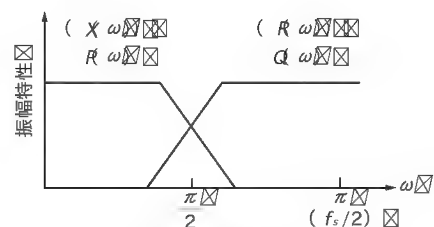
$$\hat{F}(z) = z^{-L}F(z) \quad \cdots\cdots(21)$$

したがって，（21）式が成立するためには次の二つの条件が満足されなければならない．

$$P(-z)R(z) + Q(-z)X(z) = 0 \quad \cdots\cdots(22)$$

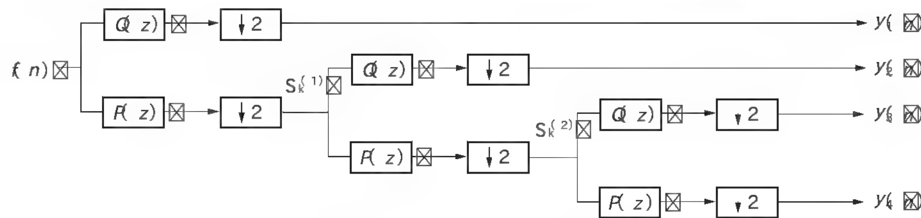


（a）2分割フィルタバンク　P，R；LPF，Q，X；HPE

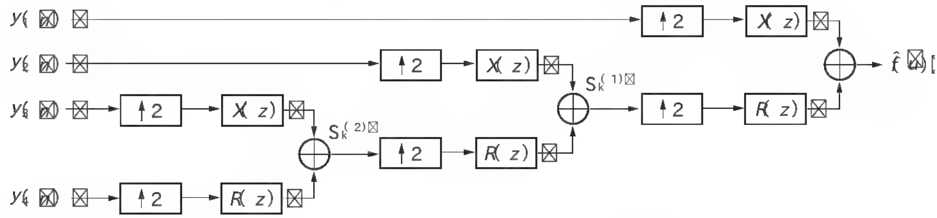


（b）周波数特性

図28　2分割フィルタ・バンク



（a）アナライザ－ウェーブレット変換



（b）シンセサイザ－逆ウェーブレット変換

図29　4チャネルウェーブレット変換対

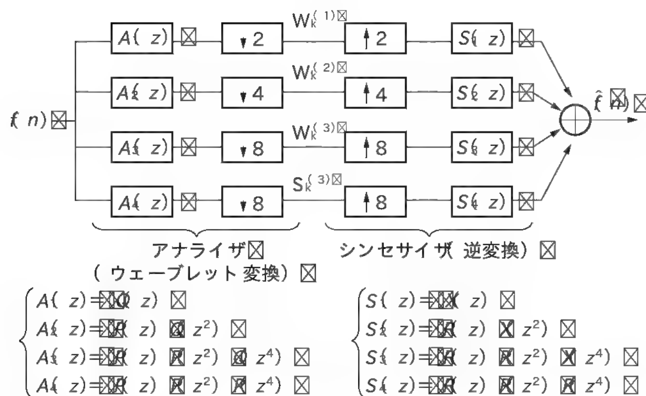


図30　図29の並列表現

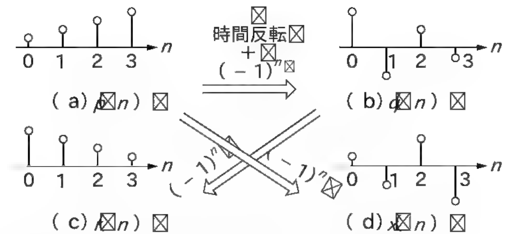


図31　CQFバンクのインパルス応答の相互関係

$$P(z)R(z)+Q(z)X(z)=2z^{-L} \quad \cdots\cdots(23)$$

この二つの条件を2分割フィルタ・バンクの設計条件という．以上よりウェーブレット変換/逆変換は，式(22)および式(23)を満たす四つのフィルタ$P(z)$，$Q(z)$，$R(z)$，$X(z)$の決定問題に帰着する．

式(22)および式(23)を満たす完全再構成フィルタ・バンクとしてCQF(Conjugate Quadrature Filter)バンクが知られている．CQFにより完全再構成になり，かつ直交したインパルス応答すなわち直交基底が得られるので直交ウェーブレット変換が求まることになる．

ここではその詳細な設計法の説明は省略するが，アナライザ$P(z)$のインパルス応答$p(n)$が設計できたとして，ほかの三つのフィルタのインパルス応答$q(n)$，$r(n)$，$x(n)$の求め方を図31に示しておく．$p(n)$の具体的な設計法は参考文献(5)が詳しい．

▶ウェーブレット変換と逆ウェーブレット変換

図29よりウェーブレット変換と逆ウェーブレット変換を数式で表現すると，以下となる[10]．

ウェーブレット変換；

$$y_k(n)=\sum_{m=-\infty}^{\infty} f(m)a_k(2^{k+1}n-m),\quad 1\le k\le M-1$$

$$y_M(n)=\sum_{m=-\infty}^{\infty} f(m)a_M(2^{M-1}n-m) \quad \cdots\cdots(24)$$

逆ウェーブレット変換；

$$f(n)=\sum_{k=1}^{M-1}\sum_{m=-\infty}^{\infty} y_k(m)S_k(n-2^{k+1}m)$$

$$+\sum_{m=-\infty}^{\infty} y_M(m)S_{M-1}(n-2^{M-1}m) \quad \cdots\cdots(25)$$

ここで，$A_k(z)$，$S_k(z)$を構成する2分割フィルタ・バンクがCQFバンクとして設計されていれば，完全再構成する直交ウェーブレット変換になる．

●演習9 信号の不連続性の検出

－時間-周波数解析－　`ex4_1_Wavelet.m`

以下の条件のウェーブレット変換を用い，いくつかの信号に対して時間周波数解析を行い，短時間フーリエ変換の結果と比較してみよう．

ウェーブレット変換：8チャネル

サンプリング周波数：512Hz

アナライザLPFインパルス応答$p(n)$は表1のとおり．

表1 アナライザLPFインパルス応答 p(n)

n	p(n)
0	3.489×10−2
1	−1.098×10−2
2	−6.286×10−2
3	0.223
4	0.556
5	0.357
6	−2.390×10−2
7	−7.594×10−2

短時間フーリエ変換と比較

窓の長さ(FFT点数) N = 32, 64, 128

窓関数：カイザー窓(MATLABコマンド：`kaiser(N, 4)`)

(1) 正弦波

$$f(n)=\begin{cases}20\sin(2\pi f_1 nT) & (0\leq n\leq 511)\\ 20\sin(2\pi f_2 nT) & (512\leq n<1023)\\ 0 & (\text{otherwise})\end{cases}$$

ただし f_1 = 50Hz, f_2 = 3Hz

解

図32～図34に示す． □

(2) 矩形パルス

$$f(n)=\begin{cases}10 & (256\leq n\leq 511)\\ -10 & (512\leq n\leq 767)\\ 0 & (\text{otherwise})\end{cases}$$

解

図35～図37に示す． □

以上の結果より，短時間フーリエ変換では，窓関数の長さにより不連続性が生じる時間が不明確になるが，ウェーブレット変換では正確に時間も周波数も特定できていることがわかる．

演習10 デノイジング `ex4_2_Wavelet.m`

つぎに，矩形パルスに以下の条件の帯域制限ノイズを加え，ウェーブレット変換を施し，デノイジング(雑音抑圧)を行ってみる．具体的には，ノイズが局在しているウェーブレット変換出力を低減し，逆ウェーブレット変換を計算して雑音を抑圧した信号を再生する．

- 平均0，分散1のガウス雑音を，アナライザ側のHPFを用いて帯域制限を行う．
- SNR = −10dB

解

図38～図41に示す．

具体的には，図40より雑音が局在している y_1 および y_2 のレベルをそれぞれ 10^{-4} 低減して雑音抑圧を行っている． □

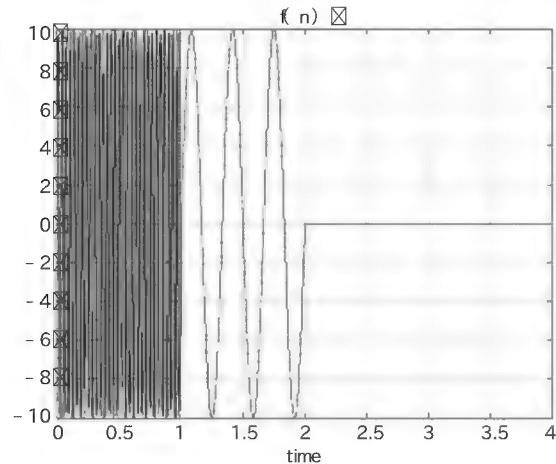


図32 入力波形 f(n)

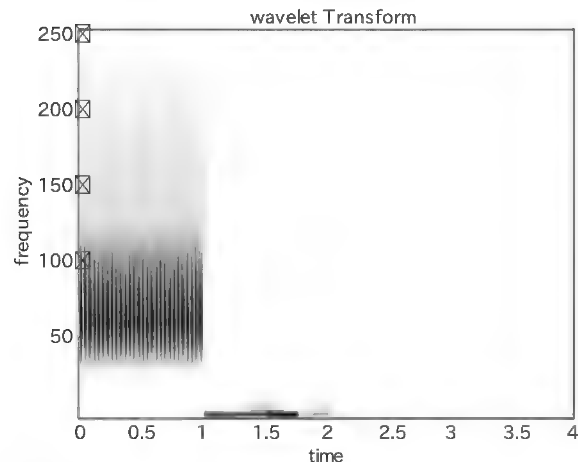


図33 ウェーブレット変換出力

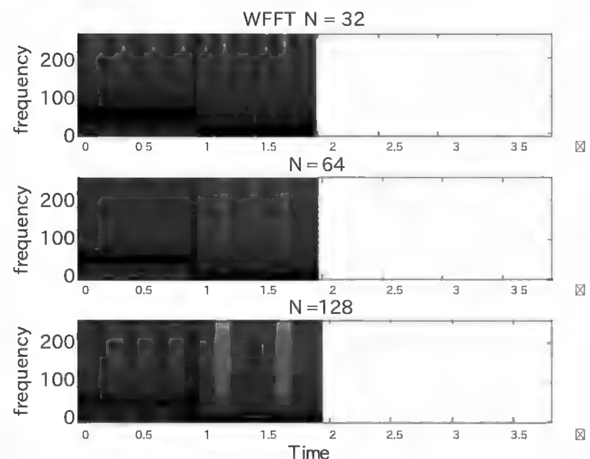


図34 短時間フーリエ変換出力(上段：N = 32，中段：N = 64，下段：N = 128)

Information ── Xilinx，新たに組み込みプロセッサ事業部門とDSP事業部門を新設

米Xilinx社は，マルチプロセッサ・コア，IPブロック，開発ツールを統合化してFPGAを使った組み込みシステムに関するサービスを提供するために，組み込みプロセッサ部門を新設した．また，DSP市場への参入を目ざしてDSP事業部門も新たに設置した．

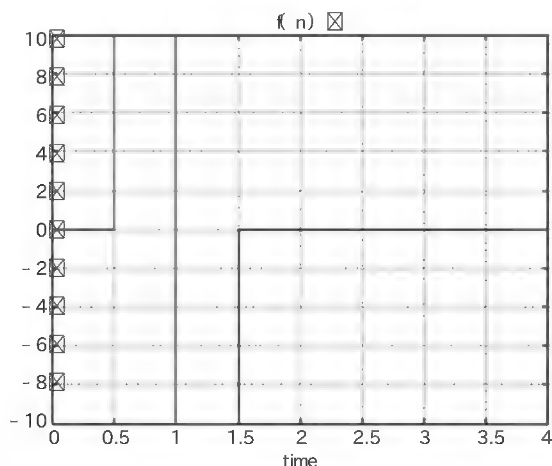


図35　入力波形 f(n)

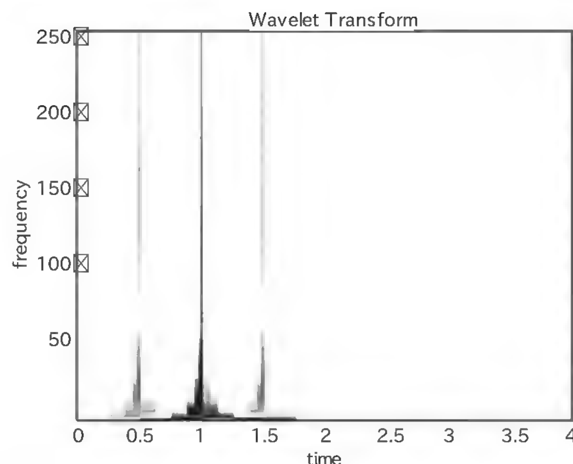


図36　ウェーブレット変換出力

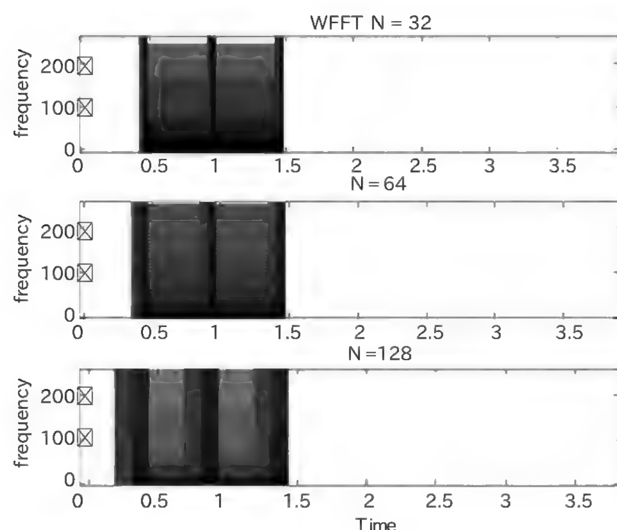


図37　短時間フーリエ変換出力（上段：$N = 32$，中段：$N = 64$，下段：$N = 128$）

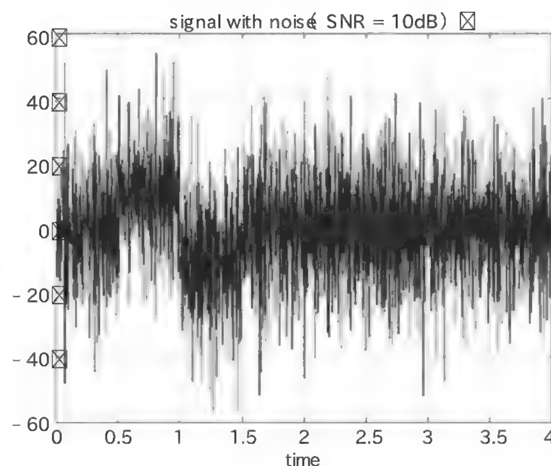


図38　ノイズが付加された信号

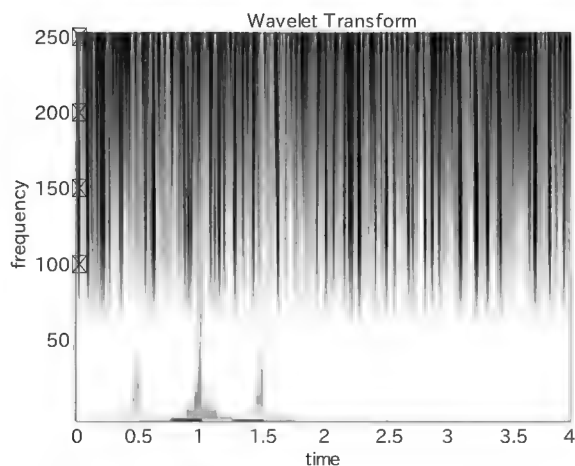


図39　ウェーブレット変換出力

このように，ウェーブレット変換により周波数に局在している雑音の抑圧が可能であり，雑音抑圧を実施してから統計処理など目的の処理を行うとよい．

おわりに

本章では，少ないデータ数で統計解析やスペクトル解析を行うテクニックをマルチレート信号処理の技術を用いていくつか紹介した．また，雑音が付加している信号に対して，ウェーブレット変換を施すことで雑音抑圧する方法も紹介した．これらのテクニック以外に，等間隔でサンプリングされていなかったり，ランダムにデータが欠損していたりするいわゆる不等間隔サンプル・データに対するサンプリング定理が最近研究されており[7]，新しい補間方法や画像圧縮技術として注目を集めている．機会を見つけて紹介していきたいと考えている．

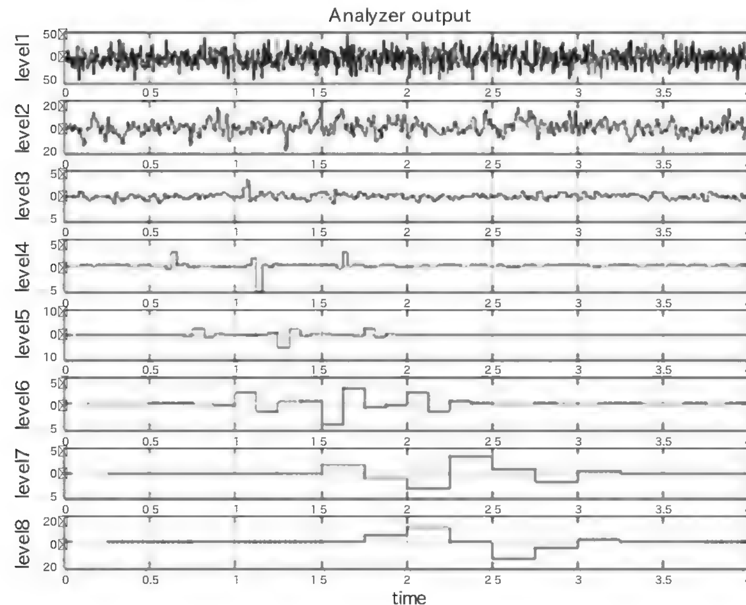


図40 アナライザ出力(デノイジング前)
Level iは図29(a)の$y_i(n)$を示す

参考文献
(1) 尾知，金城；適応フィルタ，Interface，1995年6月号，pp.175-187
(2) 篠崎，松下編；応用数値計算法入門(上)，第3章，コロナ社，1976年
(3) 秋田高専；山本昌志先生講義ノート
http://akita-nct.jp/~yamamoto/lecture/2003/5E/lecture_5E/Lagrange_Spline/Lagrange_Spline.html
(4) 高橋大輔；数値計算，岩波書店，1996年
(5) 尾知 博；シミュレーションで学ぶディジタル信号処理，CQ出版(株)，2004年
(6) 貴家仁志；マルチレート信号処理，昭晃堂，1995年
(7) F.Maarvasti，Nonuniform Sampling Theory and Practice，Kluwer Academic，2001
(8) 中野，山本，吉田；ウェーブレットによる信号処理と画像処理，共立出版，1999年
(9) 榊原 進；数理科学・ウェーブレットビギナーズガイド，東京電機大学出版局，1995年
(10) P.P.Vaidyanathan，Multirate systems and filter banks，Prentice-Hall，1993

おち・ひろし 九州工業大学情報工学部

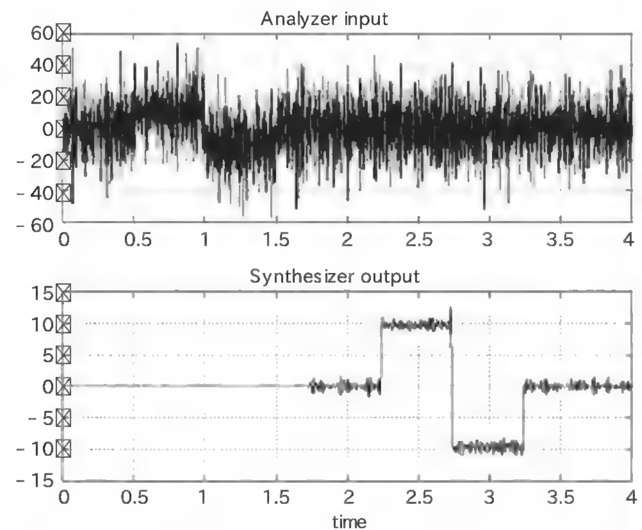


図41 デノイジング前，デノイジング後の比較(上段：デノイジング前，下段：デノイジング後)

Chapter 4

測ったデータを貯め込むことを考える

ローダブル・カーネル・モジュールを使った計測データの蓄積テクニック

実際の計測の現場において，測定したデータを安全に貯め込むことは，データを正確に取得することと同様に重要なことである．本章では，動作対象オペレーティング・システムとしてLinuxを取り上げ，ハードディスクを含む各種ストレージへの計測データの書き込み技術を中心に解説する．（筆者）

日高 亜友

データ蓄積の基本

● 測定データを貯め込むということ

測定データを貯め込む処理には，一般的に次の要件が要求されます．

●蓄積性能

データ貯め込むために必要なストレージは，計測データ量を満足させるだけのデータ容量と，単位時間あたりの計測データの取得量を満足させる，転送速度をもつ必要があります．

データ容量はストレージ・デバイスの容量でほぼ決まりますが，転送速度はシステム全体の性能が影響するので，ストレージ・デバイスの書き込み速度だけでなく，接続するバスの速度，CPU性能，キャッシュとして使用できるメモリ容量や，OSの応答性能といったさまざまな要素が関係します．

●タイム・スタンプ

データ計測には，しばしば時間軸データが必要な場合があります．時間的な要素を含むデータ自体を計測データとみなして処理する場合もありますが，一般的にはデータ蓄積時に「タイム・スタンプ」として時刻データやシステム時間を付加して蓄積するという手法を取ります．この場合，時刻データの解像度（細かさ）と，正確さが問題になります．

正確な時刻を得るためにntp（ネットワーク・タイム・プロトコル）を使用して，システム時間を合わせるといったことは心得ておくべきです．

●可搬性

計測したデータは通常，データ分析のために計測した環境からほかのシステムに移送されます．したがって，データは蓄積するだけでなく，ほかのシステムで読み出せる形式で出力できることが必要です．このデータ転送のためには，最近ではネットワークや通信を利用することが多いのですが，測定環境がネットワークの設備をもたない場合には，リムーバブルなメディアを使用して，データを移送できるように考慮する必要があります．

●安全性

計測したデータは，計測中はもちろん，計測終了後もデータ分析のためにほかの環境に移送されるまでの間，計測システム内で安全に保管されている必要があります．

たとえばRAMディスクを使用する方法を含めて，計測データをメイン・メモリに蓄積することがあります．このような場合，万が一にでも電源が切れてしまった場合，せっかく蓄積した計測データが消えてしまいます．

データ計測を行うシステムでは，このようなことのないように安全性を考慮して，目的に応じてバッテリ・バックアップやUPS（無停止電源装置），データ蓄積のために不揮発性デバイスを使用するべきでしょう．

● データ測定環境

実際に計測を行う場合には，前述の基本要件を配慮して，そのために用意した専用の環境を用います．これは専用のマシンが必要という意味ではなく，一般的なPCを使用してデータ測定を行う場合でも，その目的専用に準備した環境を構築するべきだということです．

実はデータ測定環境の準備は，各種ベンチマーク・テストの環境の構築に似ています．つまり，システム内で目的に不要な部分をなるべく切り捨てて，効率良く目的の機能を動作させるようにします．また，測定結果に影響を与えるような要素は極力排除します．

LinuxにしてもWindowsにしても，インストール直後から標準で多くの機能が有効になっています．このうち，注意しなくてはならない項目を以下に挙げます．

●自動実行プログラムの停止

cron，at，タスク・スケジューラをはじめとする自動実行プログラムは，すべて必ず停止する必要があります．

表1 ファイル・システムの種類

ファイル・システム	説 明
ext2	カーネル2.0以降のLinuxの標準的なファイル・システム
ext3	カーネル2.4以降でサポートされた，現在のLinuxで標準的なファイル・システム．ext2との互換性を持ち，ジャーナル機能によるリカバリが可能な点が特徴
reiserFS	最初からLinux用のジャーナリング・ファイル・システム*として，また小さいファイルの性能向上を目的として設計された．カーネル2.4以降で正式サポートされている
XFS	SGI社が自社のIRIXのために開発したファイル・システム．オープン・ソース化されてカーネル2.6以降で正式サポートされた・ジャーナル機能とツールが特徴で，カーネル2.4以前でも利用されていた
JFS	もともとIBMがAIXで提供していたジャーナル機能をもつ商用ファイル・システム．オープン・ソース化されてカーネル2.6で正式サポートされた
VFAT	もともとはWindows95のロング・ファイル・ネームのサポートを意味していたが，現在ではDOS互換のFATファイル・システムのことを指す場合が多い．FDなどのリムーバブル・メディアで他システムとの互換性を確保する場合には必須．テストではFAT32でフォーマットして使用した

＊：ファイル更新履歴のバックアップをとる機能をもったファイル・システム

表2 ファイル・システムのベンチマーク（RedHat 7.3 / kernel 2.4.26）

ファイル・システム	マウント・オプション	Sequential Block Write（Kバイト/s）	Sequential Block Read（Kバイト/s）
ext2	なし	50506	43115
ext2	-noatime	50567	43223
ext3	なし	42384	40617
ext3	-noatime	42979	40757
ext3 RAID0	なし	70686	68546
ext3 RAID0	-noatime	69164	68534
ext3 RAID1	なし	33638	40258
ext3 RAID1	-noatime	35331	40365
reiserfs	なし	44227	42344
reiserfs	-noatime	44541	42714
reiserfs	-nolog	44392	42958
reiserfs	-noatime, -nolog	45008	42795
reiserfs RAID1	なし	37859	42743
reiserfs RAID1	-noatime	35677	43455
reiserfs RAID1	-nolog	33822	42797
reiserfs RAID1	-noatime, -nolog	35296	43081
VFAT（FAT32）	なし	22347	39796

表3 ファイル・システムのベンチマーク（Fedora Core 2 / kernel 2.6.8）

ファイル・システム	マウント・オプション	Sequential Block Write（Kバイト/s）	Sequential Block Read（Kバイト/s）
ext2	なし	43340	34190
ext2	-noatime	43262	36417
ext3	なし	37612	39156
ext3	-noatime	38761	39307
ext3 RAID0	なし	68625	68486
ext3 RAID0	-noatime	71096	68524
ext3 RAID1	なし	32651	37066
ext3 RAID1	-noatime	30339	37370
reiserfs	なし	48016	39396
reiserfs	-noatime	46203	40672
reiserfs	-nolog	46365	41019
reiserfs	-noatime, -nolog	46096	40507
reiserfs RAID1	なし	28867	39016
reiserfs RAID1	-noatime	31639	37281
reiserfs RAID1	-nolog	29308	37107
reiserfs RAID1	-noatime, -nolog	29852	38007
VFAT（FAT32）	なし	※	

※bonnie++が異常終了したため測定していない．

●ネットワークの停止または隔離

データ収集に使用しないLAN，WANなどのネットワーク環境を停止，または遮断するか隔離します．もっとも簡単な隔離方法はテスト前にLANケーブルを抜くことです．LAN環境ではBroadcastパケットを受け取るだけで，それを処理するためにCPUが使われるので，より正確な計測を行う場合には重要になります．

●不要なサービス，デバイスの停止

まず，不要なサービス・プログラム（daemon）を停止します．サービス・プログラムの多くは，未処理時には，CPU時間を消費しないような設計になっていますが，起動させていると仮想メモリ空間をそれなりに占有しています．とくにメールやDNSなどのネットワーク関連のサービスは定期的に動作するものが多いので，計測時に使用しないのであれば停止しておきます．

また，計測に使用しない冗長なデバイスがシステムにあれば，外しておいたほうがよいでしょう．テスト中にデバイスが動作すると，CPUに割り込みが発生するということを認識しておくべきです．デバイス・ドライバは，組み込まれているだけでは使用されていないときにCPU時間は消費されません．また，ローダブル・モジュールの場合には，ロードされていなければ仮想メモリ空間の消費もないので，データ計測のためにカーネルの再構築を行う必要があるかどうかは，難しいところです．

ファイル・システムの選択とテスト

データ測定環境で見逃されがちなのがファイル・システムの選択です．計測データの発生頻度や蓄積データ量が多く，高いデータ蓄積性能が求められる場合には，とくに重要です．

Windowsでは，データベース・システムを使用しないのであれば，通常はNTFSかFATを選択します．

● Linuxのファイル・システム

Linuxのファイル・システムは種類も多く，また，さまざまなオプションがサポートされています．表1に，Linuxでサポートされているファイル・システムと簡単な説明を示します．

Linuxでは利用者の目的に合わせて自由に選択できるように，これだけの種類の利用可能なファイル・システムを用意しているので，計測データを蓄積する環境にも配慮してよいでしょう．

今回はLinuxでのデータ蓄積性能を検証する目的で，カーネル2.4で利用できる代表的なファイル・システムとオプションについて，簡単なテストによるベンチマークを行いました．表2(p.97)にその結果を示します．

また，最近ではハードディスクの価格が下がったことと，使いやすさと安定性が認められて，LinuxソフトウェアRAIDによるミラーリングも，デスクトップや組み込み環境で気軽に利用されるようになりました(コラム1)．

あわせてソフトウェアRAIDによるRAID0(ストライピング)と，RAID1(ミラーディスク)利用時の結果も表示したので比較してください．また同じマシンで，OSだけをカーネル2.6(Fedora Core 2)に入れ替えたテスト結果を表3(p.97)に示します．

● ベンチマーク・テスト

テストはPentium4/3.06GHzのマシンにRedHat 7.3をインストールして，カーネル2.4.26に入れ替えたものを使用しました．同一のパーティションをテスト前に各フォーマットでmkfsをした後，bonnie++1.05というディスク入出力性能のベンチマーク・プログラムを使用して，3回ずつテストを行い，Sequential Block ReadとSequential Block Writeの平均値を載せています．

● テスト結果の解析

テストでは，一般的に性能が上がるといわれている，`-noatime`のmountオプション(最終アクセス日付けの更新を行わない)のオプションと，ReiserFSの`-nolog`(ジャーナルを作らない)を指定して試してみましたが，これらのオプションによる顕著な差は見られませんでした．

COLUMN 1

ソフトウェアRAIDの可能性

RAIDというと専用のコントローラを必要とした高価なシステムだと思われがちでしたが，最近ではCPU性能の向上と，オペレーティング・システムで最初からサポートしているソフトウェアRAIDドライバの品質向上によって，かなり身近なものになりました．ハードウェアRAIDとソフトウェアRAIDの性能差に関しては，ここでは詳しく取り上げませんが，ソフトウェアRAIDもハードウェアRAIDと比較して，遜色ない性能が出ているという評価結果がいくつかのWebページ(Linux IDE-RAID Notes：`http://www.nobell.org/~gjm/linux/ide-raid/ide-i34raid.html`)で公開されています．

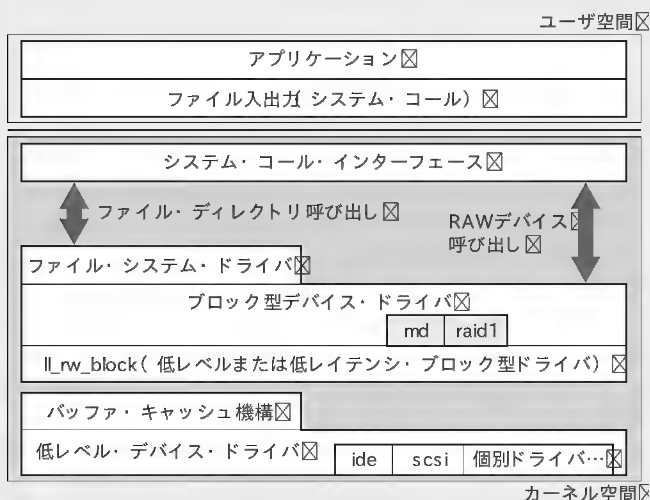
表Aに，md(LinuxソフトウェアRAID)がサポートしているRAIDの種類を示します．LinuxのソフトウェアRAIDであるmdの一番の特徴は，図Aに示すように，低レベル・ドライバと，ファイル・システム・ドライバの中間にあるデバイス・ドライバとして実装していることです．このためにLinuxの特徴である豊富なハードウェア・デバイスとファイル・システムがすべて利用可能です．

また，ソフトウェアRAIDの管理単位はパーティションごとで，しかも複数のパーソナリティ(RAIDの種類)や，ホット・スタンバイとして利用できるスペア・ディスク(実体はパーティション)を同時に利用できるので，利用目的に合わせた柔軟な使い方が可能です．

このように，LinuxでサポートしているソフトウェアRAIDドライバは使いやすく，しかも安定しているため，今後はデータ計測分野を始め，FAや組み込み分野でも安全性確保と性能向上の観点から，目的に応じて利用されるべきだと思います．

表A　mdがサポートしているRAIDの種類

パーソナリティ	説　明
リニア・モード	複数の大きさの異なるディスク(またはパーティション)を線形につなぎ合わせただけ．全体で一つのパーティションとして扱うことが可能
RAID1	いわゆるストライピング・ディスク．複数のほぼ同じ大きさのディスク(またはパーティション)を使用して複数ディスクへの読み書きを同時に行うことで性能の向上を行う
RAID2	いわゆるミラーリング・ディスク．複数のほぼ同じ大きさのディスク(またはパーティション)を使用して同一データを書き込むことで耐障害性を確保する
RAID4	3台以上のディスク(またはパーティション)を使用してブロック単位で分散させてデータをデータ・ディスクに書き，パリティ情報をパリティ・ディスクに書く方法．耐障害性と高速化の両立を狙っているが，パリティ・ディスクにアクセスが集中してボトルネックとなるため，あまり使用されない
RAID5	ブロック単位でデータとパリティ情報を分散させて書くディスクを固定しないようにして負荷配分することで，RAID4の問題点を解決した方法．3台以上のディスク(またはパーティション)を使用する場合にもっとも一般的なRAID



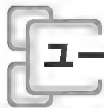
図A　mdの特徴

それよりも，選択するファイル・システムの種類による差がはっきりと現れています．ファイル・システム自体の性能の傾向としては，全般的にみるとアクセスが速い順に，

ext2 ＞ ReiserFS ＞ ext3

となります．また，RAIDに関しては，RAID0（ストライピング）で約60%前後の性能向上が見られ，RAID1（ミラーディスク）を採用すると安全性とは引き換えに，約20%前後の書き込み時の性能低下が見られます．

このように，選択するファイル・システムによって性能や特徴があり，またRAIDの組み合わせを考えると，目的に合わせてデータ蓄積に使用するファイル・システムをくふうできることがわかります．



ユーザ空間 vs カーネル空間

● 一般的なアプリケーションでの計測

計測アプリケーションでデータを測定して，その場で貯め込む場合には，一般的にリスト1のようなプログラムを書くことになります．しかし実際の計測作業においては，もう少し複雑なプログラムになり，プログラムを計測用とデータ書き込み用とで分割したり，データ書き込みをまとめて行ったり（図1），あるいはアプリケーション・レベルでの割り込みや，非同期入出力を使用する場合もあります．

これは，計測データを取得するという処理が，時間的にクリティカルな制約がある「リアルタイム的」処理である場合が多いのに対して，計測データを加工，保管するという処理は処理時間に余裕があり，後でまとめて効率良く処理する「バッチ処理的」な場合が多いからです．

リスト1　一般的な計測アプリケーション・プログラム

```
while(1) {
    if (read_data(buffer))
        write_data(buffer);
}
```

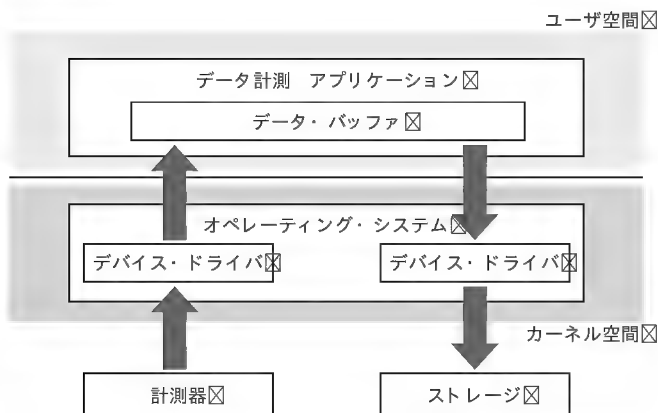


図2　アプリケーション動作のシステム内構成図

そのため，プログラム開発やメンテナンスのしやすさから処理を分割するわけです．しかし，いずれにしても「アプリケーションで計測データを読み込んで，ストレージに書き込む」という動作には変わりはありません．

このように，アプリケーションを使用してデータを計測して，その結果を保管する場合のシステム全体の構成とデータの流れを図2に示します．

● ドライバ・レベルでの計測

組み込み系の開発の経験がある方ならば，図2を見て何か改善の余地があるのではないかと，感じているのではないでしょうか．プログラム開発のしやすさよりも，処理効率の観点から，データ・バッファとデータの受け渡しをカーネル空間に移動した構成が図3です．

データの計測と蓄積ではそれぞれ別のデバイスを使用するため，別々の入出力ストリームが発生します．そして単純な計測アプリケーションの場合，アプリケーションで用意しているのはその2本のストリームを接続する単なるデータ・バッファだけということになります．そして，入出力のたびにユーザ・モードとカーネル・モードの切り替え（コンテキスト・スイッチ）とデータのコピーが発生します．このようなむだな動きを

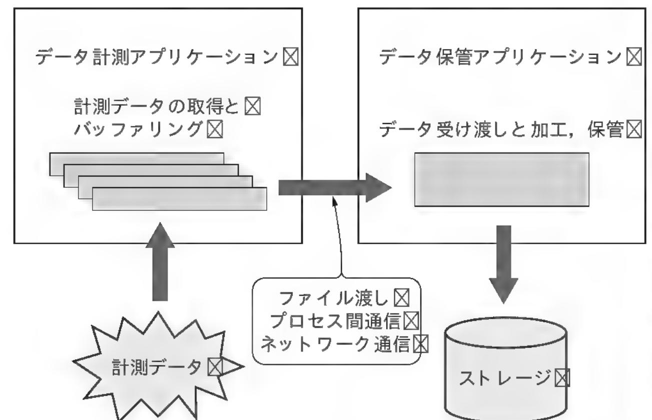


図1　データ計測用アプリケーションとデータ保管で役割分担する場合の例

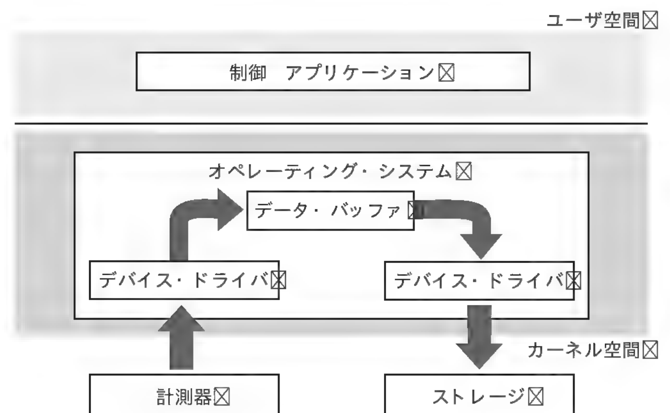


図3　ドライバ・レベルでの計測の構成

なくすために，単純で性能が要求されるような用途を想定して，ドライバ・レベルでの計測とデータ蓄積を考えてみます．

実はこのようなデータ処理をアプリケーション空間からカーネル空間に移動して，応答速度と処理効率を向上させる手法は，LinuxやUNIXの環境では古くから行われています．

たとえば，高速な処理が要求される，PPPoEやNFSでは，主要な処理をカーネル部に移したKernel Mode PPPoEやkNFSdが開発され，その後本流になりました．

また，最近では，カーネル空間に移動したWebサーバ「kHTTPd」は，httpサーバのアクセラレーション機能として，Linuxカーネルに取り込まれています．

● ドライバ・レベルでのファイル入出力ルーチン

今回はまず，ドライバ・レベルでの計測を実現させるために必要な，カーネル内で利用できる基本的なファイル入出力ルーチンを作ってみました．元のソースは前述のkHTTPdを参考にしています．リスト2にテスト用のローダブル・モジュール・ドライバのリストを示します．

テスト用モジュールは，procfsを利用して，/proc/filedataに書かれた内容を/proc/filenameで指定されたファイルに書き，/proc/filedataの読み出し時には実ファイルから読み込むといった，リダイレクタのような動作をします．

このようなローダブル・モジュールの実用性はありませんが，動作確認用のテスト・サンプルとしては十分です．open/read/write/closeの各処理を分離して作ったので，計測データの保管以外の目的にも利用できます．

● 出力テスト

デバイス・ドライバ・レベルでファイルの入出力ができることを確認したので，実際にデバイス・ドライバでデータ蓄積を行うテストを行って，アプリケーション・レベルでの蓄積と比較してみます．

まず，このルーチンの性能を調べるために，単純に内部バッファの固定長データを繰り返して，1Gバイト分のファイルを連続して書き出すテストを行い，性能を比較しました．ドライバ・モジュールを作成して，アプリケーションでのファイル出力と比較したので，その結果を表4に示します．デバイス・ドライバからのファイル出力は，アプリケーションでの出力に比べて，約20%高速になっています．

また，今回のルーチンは，カーネル空間の中でもユーザ呼び出しがカーネル内に入った直後に呼び出されるサービスをフックしているので，カーネル内のバッファ・キャッシュが有効で，カーネルでサポートされているすべてのファイル・システムと，ストレージ・デバイスやソフトウェアRAIDなどを利用するこ

表4　1Gバイト・ファイルの連続書き出し性能の比較

	Write(ms)	Write(Mバイト/s)
アプリケーション	22827	45.35
デバイス・ドライバ	18700	56.74

リスト2　テスト用のローダブル・モジュール・ドライバ(drvfile.c)

```
#ifndef __KERNEL__
#  define __KERNEL__
#endif
#ifndef MODULE
#  define MODULE
#endif

#include <linux/config.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/string.h>
#include <linux/proc_fs.h>
#include <linux/file.h>
#include <asm/uaccess.h>
#include <asm/page.h>

#define BUFFER_SIZE 2048
#if (PAGE_SIZE < BUF_SIZE)
#define BUFFER_SIZE PAGE_SIZE
                             /* for not enough page size system */
#endif

static char filename[BUFFER_SIZE];
                                     /* should not include '¥n' */
static char filedata[BUFFER_SIZE]; /* should include '¥n' */

/*
 * fs file open / read / write / close
 */
#define file_err(format, arg...)  printk(KERN_ERR "%s : "
                              format "¥n", __FUNCTION__, ## arg)
#ifdef DEBUG
#define file_dbg(format, arg...)  printk(KERN_ERR "%s : "
                              format "¥n", __FUNCTION__, ## arg)
#else
#define file_dbg(format, arg...) do {} while (0)
#endif

struct file *file_open(char *filename, int flags, int mode)
{
    struct file *filp;

    filp = filp_open(filename, flags, mode);
    if (filp==NULL || IS_ERR(filp)) {
        file_err("cannot open file = %s", filename);
        return NULL;  /* Or do something else */
    }
    if (filp->f_op->read == NULL || filp->f_op->write == NULL) {
        file_err("File(system) doesn't allow reads / writes");
        return NULL;
    }
    if (!S_ISREG(filp->f_dentry->d_inode->i_mode)) {
        filp_close(filp, NULL);
        file_err("%s is NOT a regular file", filename);
        return NULL;  /* Or do something else */
    }
    file_dbg("file mode = %08x, f_pos = %d",
         filp->f_dentry->d_inode->i_mode, (int) filp->f_pos);
    return(filp);
}

int file_read(struct file *filp, void *buf, int count)
{
    mm_segment_t oldfs;
    int BytesRead;

    oldfs = get_fs();
    set_fs(KERNEL_DS);
    BytesRead = filp->f_op->read(filp, buf, count, &filp->f_pos);

    file_dbg("BytesRead = %d", BytesRead);

    set_fs(oldfs);
```

リスト 2　テスト用のローダブル・モジュール・ドライバ（drvfile.c）（つづき）

```
    return BytesRead;
}

int file_write(struct file *filp, void *buf, int count)
{
    mm_segment_t oldfs;
    int BytesWrite;

        oldfs = get_fs();
        set_fs(KERNEL_DS);
    /* filp->f_pos = StartPos; */
        BytesWrite = filp->f_op->write(filp, buf, count,
                                          &filp->f_pos);

    file_dbg("BytesWrite = %d", BytesWrite);

        set_fs(oldfs);
    return BytesWrite;
}

void file_close(struct file *filp)
{
    fput(filp);
    filp_close(filp, NULL);
}

/*
 * read_proc -- called when user reading
 */
int filename_read(char *buf, char **start, off_t offset,
                  int count, int *eof, void *data)
{
    char in_data[BUFFER_SIZE];
    int return_length = 0;

    file_dbg("count = %d, off = %d", count, (int) offset);
    if (offset == 0) {
         memset(in_data, 0, BUFFER_SIZE);
         return_length = sprintf(in_data, "%s¥n", filename);
         memcpy(buf, in_data, return_length);
         file_dbg("filename = [%s]", filename);
    }
    *eof = 1;
    *start = buf + offset;
    return return_length;
}

int filedata_read(char *buf, char **start, off_t offset,
                  int count, int *eof, void *data)
{
    struct file *filp;
    char in_data[BUFFER_SIZE];
    int return_length = 0;
    int len = 0;

    file_dbg("count = %d, off = %d", count, (int) offset);
    if (offset == 0
        && (filp = file_open(filename, O_RDONLY, 0)) != NULL) {
        len = file_read(filp, filedata, BUFFER_SIZE);
        file_close(filp);

        memset(in_data, 0, BUFFER_SIZE);
        return_length = sprintf(in_data, "%s", filedata);
        memcpy(buf, in_data, return_length);
        file_dbg("filedata = [%s], size = %d", filedata, len);
    }
    *eof = 1;
    *start = buf + offset;
    return return_length;
}

/*
 * write_proc -- called when user writing
 */
int filename_write(struct file *file, const char *buf,
                                unsigned long count, void *data)
{
    int length;

    file_dbg("count = %d", (int) count);
    length = count > BUFFER_SIZE ? BUFFER_SIZE : count;
                                       /* limit of max size */
    copy_from_user(filename, buf, length);
    if (filename[length-1] == '¥n') /* Is last char LF? */
        filename[length-1] = '¥0';
    filename[length] = '¥0';

    file_dbg("finename = [%s], size = %d", filename, length);

    return length;
}

int filedata_write(struct file *file, const char *buf,
                unsigned long count, void *data)
{
    struct file *filp;
    int length;
    int len = 0;

    file_dbg("count = %d", (int) count);
    length = count > BUFFER_SIZE ? BUFFER_SIZE : count;
                                        /* limit of max size */

    copy_from_user(filedata, buf, length);
    filedata[length] = '¥0';

    filp = file_open(filename, O_CREAT|O_WRONLY|O_TRUNC, 0666);
    if (filp != NULL) {
        len = file_write(filp, filedata, length);
        file_close(filp);

        file_dbg("writedata = [%s], size = %d,
                wrote = %d", filedata, length, len);
    }
    return length;
}

/*
 * top level routine
 */
static void file_create_proc(void)
{
    struct proc_dir_entry *entry;

    strcpy(filename, "/tmp/testfile");
    strcpy(filedata, "This is a test data.¥n");

    entry = create_proc_entry("filename", 0, 0);
                            /* "filename" registration */
    entry->read_proc = filename_read;   /* read routine */
    entry->write_proc = filename_write; /* write routine */

    entry = create_proc_entry("filedata", 0, 0);
                            /* "filedata" registration */
    entry->read_proc = filedata_read;   /* read routine */
    entry->write_proc = filedata_write; /* write routine */
}

static void file_remove_proc(void)
{
    remove_proc_entry("filedata", NULL);
    remove_proc_entry("filename", NULL);
}

int __init file_init_module(void)
{
    file_create_proc();
    file_dbg("init file");
    return 0;
}

void __exit file_cleanup_module(void)
{
    file_dbg("cleanup file");
    file_remove_proc();
}

MODULE_DESCRIPTION("FILE IO from kernel space");
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Device Drivers Limited");

module_init(file_init_module);
module_exit(file_cleanup_module);
```

とが可能です．

● タイム・スタンプとJiffies

この計測アプリケーションでは，ファイルの出力による時間を計測するために，gettimeofday()を使用して計測に必要な時間データを取得しています．

gettimeofday()を使用すると，原理的にはμsまでの単位で現在時刻が取得できるのですが，Linuxはマルチユーザによる TSS 環境であるために，厳密にユーザが希望した瞬間に目的の時間情報が取得できるとは限らず，多少のずれが発生する場合があることを理解しておく必要があります．

一方で，テスト用ドライバでは，カーネル内での標準タイマである jiffiesを用いて時間を測定しています．

jiffies値はその名のとおりカーネル内において，その瞬間のタイム・スライス値を示すグローバル変数になっているため，特別なルーチンを呼び出さなくとも，すべてのカーネル内ルーチンから参照可能になっています．

jiffies値は，システムの起動時に0にリセットされ，1タイム・スライス(カーネル2.4の標準は10ms)に1ずつ増えていきます．実はgettimeofday()も粒度単位は違うものの，元々はjiffiesによって経過時間を取得しています．1sと1タイム・スライス(jiffies時間)の関係を定義しているのが，HZ定数マクロです．1s間をいくつで割った値を1タイム・スライスとするかがHZなので，カーネル2.4のHZは通常100になっています．

なお，カーネル2.4のjiffies値は，32ビット(unsigned long)なので，約1.3年(496日)で溢れるということに注意が必要です．

● カーネル2.6のJiffies

jiffies値の取り扱いは，カーネル2.6で変更になりました．まず変数が従来の32ビットから64ビットに拡張されて，長期間運用によるオーバフローを考慮する必要がなくなりました．

また，デフォルトのHZ値が，従来の100から1000に変わり

COLUMN 2

CPUのRDTSC命令

AMD Athlonも含めたPentium以降のx86系CPUには，RDTSCという命令があります．

この命令は，起動時に0から始まり，1クロックごとにカウント・アップする，CPU内の64ビット・カウンタ(TSC)を読み出します．C言語から呼び出す命令やライブラリはないので，リストAに示すように，asmマクロを用いてコーディングします．RDTSCの実行はユーザ・モードで可能なので，すべてのアプリケーションからオーバヘッドなしで利用できます．

いわば，ハードウェアで実現したjiffiesのようなものですが，読み出す単位はあくまでもクロックなので，システムによって，単位時間に関する互換性(当然ながらCPUのクロックに依存する)と，返す値の互換性がないという点に注意が必要です．たとえばAthlonXPでは，同じクラスのPentium4と比べて約半分かそれ以下の値を返します．

Linuxカーネル・ツリーでは，include/asm/msr.hにRDTSCを使用したマクロが定義されている(リストB)ので，それを利用すると便利です．このマクロを使用したコーディング例をリストCに示します．

本題からは外れますが，このRDTSC命令を使うと，アプリケーションやドライバで，指定した区間の「実際の実行クロック数」がわかるので，簡易的なプロファイラとして利用することができます．

リストA　rdtsc.c

```
#include <stdio.h>

inline unsigned long long int rdtsc()
{
        unsigned long long int x;
        __asm__ volatile ("rdtsc" : "=A" (x));
        return x;
}

void something(void)
{
}

int
main()
{
        long long a, b;

        a = rdtsc();
        something();
        b = rdtsc();

        printf("clock = %ld¥n", (long) (b - a));
        return 0;
}
```

リストB　asm/msr.hに記述のrdtscマクロ

```
#define rdtsc(low,high) ¥
        __asm__ __volatile__("rdtsc" : "=a" (low), "=d" (high))
#define rdtscl(low) ¥
          __asm__ __volatile__("rdtsc" : "=a" (low) : : "edx")
#define rdtscll(val) ¥
                    __asm__ __volatile__("rdtsc" : "=A" (val))
```

リストC　ドライバ内でのrdtscマクロの使い方

```
{
    long long a, b;

      rdtscll(a);
      something();
      rdtscll(b);
      printk("**rclock = %ld¥n", (long) (b-a));
}
```

ました．これはプロセス・スケジューリングの単位であるタイム・スライスが，カーネル2.6では従来の10msから1msに変わったということも同時に示しています．

ここ数年間のCPUクロックの向上を考えると，このタイム・スライス値の変更は妥当なように思えます．

● 処理の実際

次に実際にデータを取得して，取得したデータをハードディスクのファイルに収める処理で比較します．今回は，高速大容量のデータ取得をシミュレーションするために，100Base-TのLANを使用することにしました．

TCP/IPをそのまま使用したのでは，プロトコル処理のオーバヘッドがあるので，テストで使用するアプリケーション・プログラムは，送信側，受信側ともにRAWSOCKTを使用しました．送信側は，通信相手のACKを確認せずにパケットを連続して送りつづけます．受信側は，受信したパケットをすべて無条件にディスクに保管するようにしてテストをしました．アプリケーションのソースは誌面のつごうにより割愛します．

デバイス・ドライバでの実装は，「Embedded UNIX Vol.4」に掲載されている「基礎からのデバイスドライバ作成講座（第2回）」の「パケットフィルタ・ドライバの解説，作成と応用」で解説している，汎用入力パケット・フィルタを改造して，やはり取得したパケットをすべてディスクに保管するようにしました．

実はパケット・フィルタ部分は割り込みモードで動作するので，単純にフィルタ・ドライバ内からファイルI/Oルーチンを呼び出して動作させることはできないことから，コーディングとデバッグには少々苦労します．今回はコラム3に掲載している，カーネル・スレッドをほとんどそのまま使用して，受信したパケットをフィルタ・ドライバからリング・バッファを経由してデータ書き込み専用のカーネル・スレッドに受け渡すようにして，スムーズに動作させています．おもな変更点は，ソース中の`kth_do_sleep()`の部分で，スリープせずにリング・バッファからパケットをなくなるまで取り出して，ファイルに書くようにするだけです．

● パケット受信テスト

送信側から，1000バイトのTCP/IPパケットを連続して送信し，アプリケーション・プログラムと，デバイス・ドライバとで受信テストを行った結果を表5に示します．このテストでは，

表5 受信性能のテスト結果

送信パケット数	10,000	100,000	1,000,000
アプリケーション受信パケット数	10,000	95,734	741,959
デバイス・ドライバ受信パケット数	10,000	100,000	999,984

※パケットは1000バイト，3回テストした平均値．

処理速度はほぼ送信側のマシンの性能に依存するので，どれぐらいパケットを取りこぼしなく取得して，蓄積できたのかを比較しています．

デバイス・ドライバは，アプリケーションでの受信処理と比較して，ほとんど取りこぼしがないのを確認できました．

このテストでは，送信側はPentiumIII 1.26GHzの最近ではやや遅いマシンを使用し，受信側は，Pentium4 3.06GHz RedHat 7.3にKernel 2.4.26を入れて，ext3でオプションなしにマウントした同じパーティションにデータを保管して，3回テストした平均値を載せています．

おわりに

このように，計測したデータを蓄積するために，オペレーティング・システムが提供しているさまざまな方法を組み合わせて，より安全に，より確実に目的を達成させるためのいくつかの手法を紹介しました．

実務での利用を考慮して，最新版のソース・コードは弊社のダウンロード・ページ（http://www.devdrv.co.jp/）でも近いうちに公開していきます．

参考文献
（1）Embedded Unix，vol. 4. 基礎からのデバイスドライバ作成講座（第2回），CQ出版（株）
（2）The Software-RAID HOWTO
http://www.linux.or.jp/JF/JFdocs/The-Software-RAID-HOWTO.html
（3）Linuxのファイルシステムとファイルの概要
http://japan.linux.com/kernel/03/10/14/0235259.shtml
（4）Linux IDE-RAID Notes
http://www.nobell.org/~gjm/linux/ide-raid/ide-i34raid.html

ひだか・あとむ （株）デバイスドライバーズ

COLUMN 3

SMP システムとカーネル・スレッド

すでにご存じのように，動作クロック・アップによる CPU の性能向上にかげりが見え始めたことから，マルチ CPU やマルチコアを取り入れた新しいアーキテクチャの CPU が登場しつつあります．また，シングル CPU であっても，パイプラインの隙間で，別のスレッドを実行させる Intel の HyperThreading Technology のような SMT(Simultaneous Multithreading Technology)も，Pentium4の普及のおかげで一般的になっています．

Linux をはじめとするマルチタスク環境のアプリケーションの開発では，多くの場合 POSIX 準拠のスレッドがサポートされているので，ただ線形に単一プログラムの実行性能を向上させるだけでなく，複数の処理を多重化させることによる性能向上でも，比較的低いコスト(労力や処理のオーバヘッド)で恩恵を受けることができます．

それでは，今回紹介したようなカーネル・モードでのデータ処理やデバイス・ドライバにおいて，マルチスレッドを実現して，「パケットの読み出しとデータの書き出し」を別々の CPU で同時実行させるような手段による，処理効率の向上は可能なのでしょうか．実は Linux では，カーネル自体はマルチスレッドを使用してコーディングされているため，その解説はあるのですが，カーネル空間で動作するデバイス・ドライバやローダブル・モジュールを新たに開発する際に，自由にマルチスレッド・プログラミングを行う方法については，あまり説明されていませんでした．

このようなカーネル空間で生成されて，スケジューリングの単位としてタイム・スライスが割り当てられていながら，決してユーザ・モードにならないスレッドを Kernel Thread と呼びます．この Kernel Thread をうまく自由に扱えるようになると，今後到来するマルチ CPU 時代に効率良く動作するデバイス・ドライバや，カーネル・モジュールを書くことができそうです．

今回は，カーネルのソース中で比較的わかりやすい，ソフトウェア RAID ドライバで複数のディスク・データの同期を行う際のスレッド処理を参考にして，Kernel Thread を用いたプログラミングを行ってテストしてみました．

まず，Kernel Thread 部分だけを実行するモジュール「kthread.c」をテストして，その後本文中で紹介した，パケット受信とディスク保管を行うデバイス・ドライバに組み込みました．

(Hyperthread ではない，本当の)SMP マシンで実験したのですが，スレッドで分割したパケットを受信する処理と，受信パケットをディスクに書く処理は，二つの CPU で完全に読み込みと書き込みで分担するようには動作しませんでした．しかし，当然ながらほかの負荷をかけた状態では，同じクロックのシングル CPU (UP)と比べて明らかにパケットの取りこぼしが減っています．表 5 に示したように，今回の例ではもともとドライバの処理性能が良いため，負荷がない状態では，UP/SMP での顕著な差は出ませんでした．また，Hyperthread の ON/OFF も試したのですが，やはり差が出ませんでした．

カーネル・スレッドはそのほかに，本文中で紹介したように，カーネル内のプログラミングで，割り込み処理ルーチンやスケジュールで動作できない処理を実行させることや，daemon サービスのように定期的に動作させる目的にも利用できます．

ここで紹介したカーネル・スレッド実行中の ps 画面を図 B に，「kthread.c」をリスト D に示します．

図 B　ps コマンドでの Kernel Thread 実行中の確認画面

```
  PID TTY      STAT   TIME COMMAND
    1 ?        S      0:04 init [3]
    2 ?        SW     0:00 [keventd]
    3 ?        SW     0:00 [kapmd]
    4 ?        SWN    0:00 [ksoftirqd_CPU0]
    5 ?        SW     0:48 [kswapd]
    6 ?        SW     0:32 [bdflush]
    7 ?        SW     0:00 [kupdated]
    8 ?        SWN    0:10 [mdrecoveryd]
   13 ?        SW<    0:04 [raid1d]
   14 ?        SWN    0:11 [raid1syncd]
   15 ?        SW<    0:00 [raid1d]
   16 ?        SW     0:00 [kjournald]
  136 ?        SW     0:00 [kjournald]
  137 ?        SW     0:00 [kjournald]
  138 ?        SW     0:00 [kjournald]
  416 ?        S      0:00 syslogd -m 0
  421 ?        S      0:00 klogd -x
  519 ?        S      0:00 /usr/sbin/sshd
  540 ?        S      0:00 xinetd -stayalive -reuse
                           -pidfile /var/run/xinetd.pid
  552 ?        S      0:00 gpm -t ps/2 -m /dev/mouse
  604 ?        S      0:00 xfs -droppriv -daemon
  611 ?        S      0:00 login -- root
  612 tty2     S      0:00 /sbin/mingetty tty2
  613 tty3     S      0:00 /sbin/mingetty tty3
  614 tty4     S      0:00 /sbin/mingetty tty4
  615 tty5     S      0:00 /sbin/mingetty tty5
  616 tty6     S      0:00 /sbin/mingetty tty6
  617 ?        S      0:00 login -- root
  620 ttyS0    S      0:00 -bash
  718 ?        S      0:00 /usr/sbin/sshd
  719 pts/1    S      0:00 -bash
  751 pts/1    S      0:00 su
  752 pts/1    S      0:00 bash
  777 ?        S      0:00 /usr/sbin/sshd
  778 pts/2    S      0:00 -bash
  810 pts/2    S      0:00 su
  811 pts/2    S      0:00 bash
  935 tty1     S      0:00 -bash
 2540 ?        SW     0:00 [kth_sleep]
 2555 pts/1    R      0:00 ps ax
```

今回作成した Kernel Thread

リスト D　kthread.c

```
#ifndef __KERNEL__
#  define __KERNEL__
#endif
#ifndef MODULE
#  define MODULE
#endif

#include <linux/config.h>
#include <linux/module.h>
#include <linux/vmalloc.h>
```

リスト D　kthread.c(つづき)

```
#include <linux/kernel.h>
#include <linux/sysctl.h>
#include <linux/init.h>
#include <linux/kmod.h>
#include <linux/unistd.h>
#include <linux/string.h>
#include <linux/fs.h>
#include <linux/proc_fs.h>
#include <asm/uaccess.h>
#include <asm/page.h>
#include<linux/mm.h>
#include<asm/semaphore.h>
#include<linux/ioctl.h>
#include<linux/types.h>
#include<linux/seq_file.h>
#include<linux/smp_lock.h>
#include<linux/delay.h>
#include<linux/locks.h>
#include<linux/kernel_stat.h>
#include<linux/completion.h>
#include<asm/io.h>
#include<asm/bitops.h>

#define BUFFER_SIZE 12
#if (PAGE_SIZE < BUF_SIZE)
#define BUFFER_SIZE PAGE_SIZE
                        /* for not enough page size system */
#endif

static unsigned long kth_cmd = 0;
static int debug;

DECLARE_WAIT_QUEUE_HEAD (kth_wait);

static struct clientdata {
    unsigned long jiffies;
    task_queue *queue;
} kth_data;

static struct timer_list kth_timer;

static void kth_timedout(unsigned long ptr)
{
    if (debug) {
         printk("** kth_timedout¥n");
    }
        wake_up_interruptible(&kth_wait);
                                     /* awake the process */
}

static unsigned int kth_sleep(unsigned long ms)
{
    if (debug) {
         printk("** kth_sleep strat¥n");
    }
        kth_data.jiffies = jiffies;
        kth_data.queue = NULL; /* don't requeue */

        init_timer(&kth_timer);/* init the timer structure */
        kth_timer.function = kth_timedout;

        printk("**init_timer & timer_function done =
                                           %ld ms¥n", ms);

        kth_timer.data = (unsigned long)&kth_data;
        /* kth_timer.expires = jiffies + HZ; * one second */
        kth_timer.expires = jiffies + (ms / 10);
                                        /* ms millisecond */

        add_timer(&kth_timer);
        interruptible_sleep_on(&kth_wait);
        del_timer_sync(&kth_timer);
                          /* in case a signal woke us up */

    if (debug) {
         printk("** end of kth_sleep **¥n");
    }
        return jiffies;
}

/*
 * kernel thread
 */
typedef struct kth_thread_s {
        void                    (*run) (void *data);
        void                    *data;
        wait_queue_head_t    wqueue;
        unsigned long            flags;
        struct completion        *event;
        struct task_struct       *tsk;
        const char               *name;
} kth_thread_t;

static inline void kth_init_signals (void)
{
        current->exit_signal = SIGCHLD;
        siginitsetinv(&current->blocked, sigmask(SIGKILL));
}

static inline void kth_flush_signals (void)
{
        spin_lock(&current->sigmask_lock);
        flush_signals(current);
        spin_unlock(&current->sigmask_lock);
}

static kth_thread_t *kth_sleep_thread;

#define THREAD_WAKEUP  0

/*
 * thread main
 */
void kth_do_sleep(void *data)
{
    int i;
    long sec = (long) data;

        printk("*** KTH sleep thread got woken up ...¥n");

    for(i = 1; i < 10; i++) {
         printk("** %d: thread ON, sec = %ld¥n", i, sec);
         kth_sleep(sec * 1000);
    }
        printk("*** KTH sleep thread finished ...¥n");
}

int kth_thread(void *arg)
{
        kth_thread_t *thread = arg;

        lock_kernel();
        daemonize();
        sprintf(current->comm, thread->name);
        kth_init_signals();
        kth_flush_signals();

        thread->tsk = current;
        current->policy = SCHED_OTHER;
        /* current->nice = -20; */
        unlock_kernel();
        complete(thread->event);

        while (thread->run) {
                void (*run)(void *data);
                DECLARE_WAITQUEUE(wait, current);

                add_wait_queue(&thread->wqueue, &wait);
                set_task_state(current, TASK_INTERRUPTIBLE);
                if (!test_bit(THREAD_WAKEUP, &thread->flags))
{
                        printk("*** thread %p went to
                                     sleep.¥n", thread);
                        schedule();
                        printk("*** thread %p woke up.¥n",
                                              thread);
                }
                current->state = TASK_RUNNING;
                remove_wait_queue(&thread->wqueue, &wait);
                clear_bit(THREAD_WAKEUP, &thread->flags);

                run = thread->run;
                if (run) {
                        run(thread->data);
                        run_task_queue(&tq_disk);
                }
                if (signal_pending(current))
                        kth_flush_signals();
        }
        complete(thread->event);

    return(0);
```

リスト D　kthread.c（つづき）

```
}

/*
 * support routines
 */
void kth_wakeup_thread(kth_thread_t *thread)
{
        printk("** waking up KTH thread %p.¥n", thread);
        set_bit(THREAD_WAKEUP, &thread->flags);
        wake_up(&thread->wqueue);
}

kth_thread_t *kth_register_thread(void (*run) (void *),
                               void *data, const char *name)
{
    int ret;
    kth_thread_t *thread;
        struct completion event;

        thread = (kth_thread_t *)
        kmalloc(sizeof(kth_thread_t), GFP_KERNEL);
        if (!thread)
                return NULL;

        memset(thread, 0, sizeof(kth_thread_t));
        init_waitqueue_head(&thread->wqueue);

        init_completion(&event);
        thread->event=&event;
        thread->run=run;
        thread->data=data;
        thread->name=name;

        ret = kernel_thread(kth_thread, thread, 0);
        if (ret < 0) {
                kfree(thread);
                return NULL;
        }

        wait_for_completion(&event);
    return(thread);
}

void kth_interrupt_thread(kth_thread_t *thread)
{
        if (!thread->tsk) {
                printk("** interrupt error¥n");
                return;
        }
        printk("** interrupting KTH-thread pid %d¥n",
                                          thread->tsk->pid);
        send_sig(SIGKILL, thread->tsk, 1);
}

void kth_unregister_thread(kth_thread_t *thread)
{
        struct completion event;

        init_completion(&event);

        thread->event = &event;
        thread->run = NULL;
        thread->name = NULL;
        kth_interrupt_thread(thread);
        wait_for_completion(&event);

        kfree(thread);
}

/*
 * kth_read_proc -- called when user reading
 */
int kth_read_proc(char *buf, char **start, off_t offset,
                            int count, int *eof, void *data)
{
    char in_data[BUFFER_SIZE];
    unsigned long req_cmd = 0;
    int return_length = 0;

    printk("**read_proc(), count = %d, off = %d¥n",
           count, (int) offset);
    if (offset == 0) {

         req_cmd = kth_cmd;

         if (req_cmd > 0 && req_cmd < 20) {
              (void) kth_sleep(req_cmd * 1000);
         }
         memset(in_data, 0, BUFFER_SIZE);
         return_length = sprintf(in_data, "%lu¥n", req_cmd);
         memcpy(buf, in_data, return_length);
         printk("**read_proc(), cmd = %lu¥n",
                          simple_strtoul(in_data, NULL, 10));
    }

    kth_interrupt_thread(kth_sleep_thread);

    *eof = 1;
    *start = buf + offset;
    return return_length;
}

/*
 * kth_write_proc -- called when user writing
 */
int kth_write_proc(struct file *file, const char *buf,
                            unsigned long count, void *data)
{
    char out_data[BUFFER_SIZE];
    int length;
    char *p;

    printk("**write_proc(), count = %d¥n", (int) count);

    length = count > BUFFER_SIZE ? BUFFER_SIZE : count; /*
                                          limit of max size */

    copy_from_user(out_data, buf, length);

    p = &out_data[0];
    kth_cmd = simple_strtoul(p, &p, 10);

    printk("write_file(), cmd = %ld¥n",
           kth_cmd);

    kth_wakeup_thread(kth_sleep_thread);

    return length;
}

/*
 *
 */
static void kth_create_proc(void)
{
    struct proc_dir_entry *entry;
    entry = create_proc_entry("kthread", 0, 0);
    entry->read_proc = kth_read_proc; /* read routine */
    entry->write_proc = kth_write_proc; /* write routine */

    kth_sleep_thread = kth_register_thread(kth_do_sleep,
                                  (void *) 5, "kth_sleep");
}

static void kth_remove_proc(void)
{
    kth_unregister_thread(kth_sleep_thread);

    remove_proc_entry("kthread", NULL);
}

int __init kth_init_module(void)
{
    kth_create_proc();
    printk("<1>KTH, init¥n");
    return 0;
}

void __exit kth_cleanup_module(void)
{
    printk("<1>KTH cleanup¥n");
    kth_remove_proc();
}

MODULE_DESCRIPTION("KTH Module");
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Device Drivers Limited");
MODULE_PARM(debug,      "i");
MODULE_PARM_DESC(debug, "KTH debug level (0-6)");

module_init(kth_init_module);
module_exit(kth_cleanup_module);
```

Chapter 5

ピックアップ・コイルをセンサとした位置データ計測による

交流磁界を用いたモーション・キャプチャの開発

本章では，データ計測の応用例として，磁気センサを用いた位置の割り出し，すなわち「モーション・キャプチャ」の製作例を示す．磁界の中に存在するコイルの位置を計測し，最終的に「バーチャルはえたたき」と「バーチャル物体搬送」を実現する．（編集部）

熊谷 正朗

情報やロボットに関わる分野では，その機器が取り扱う情報量が複雑に，そして膨大になるに従って，いかに人間と情報をやり取りするかが重要になってきました．機器から人間に提示するための一つの方法は，立体視による3次元の視覚情報の提示です．通常の画像は2次元の情報しかもちませんが 注1，特殊なレンズを貼り付けたディスプレイや，頭部に装着するヘッド・マウント・ディスプレイ（HMD）を使用することで，左右の目に独立した画像を提示して立体感を感じさせることができます．一方，人間から機器に情報を入力するためにもさまざまな手法が開発されてきました．とくに人間の動作を3次元情報で取得するために，モーション・キャプチャと呼ばれる装置がいろいろと開発されています．ロボットの遠隔操作に使用されたり，人間の運動情報の取得（医療，体育，伝統芸能保存，ゲームなど）に使われています．

筆者が所属していた研究室でもバーチャル・リアリティの研究を行っており，その一環として，対象の位置と姿勢を検出できる，磁気を用いたモーション・キャプチャ装置の開発が行われました．

本章では，筆者が当初の原理[1]を改良しつつ[2]，ハードウェアによる高速ディジタル信号処理などを導入することで開発した，実用化プロトタイプについて，その原理と実装の詳細を解説します．

位置検出の原理

● 交流磁界とピックアップ・コイル

本手法を要約すると，以下のようになります．

- 計測対象空間を囲む励磁コイルに交流電流を流して，計測空間に交流磁界を発生させる
- 交流磁界により，計測対象点に取り付けたピックアップ・コイルに電圧が誘起する
- 生じた電圧から励磁した成分の振幅を取り出し，処理する

ここで，ピックアップ・コイルに誘起する信号の振幅は，その地点での磁界とピックアップ・コイルの方向によって決まります．そのため，空間に姿勢の基準となるような磁界と，位置の基準となるような磁界をつくれば検出ができます．前者のためには，なるべく空間内で均一で一定方向を向くような磁界が，後者のためには，位置によってなるべくリニアに変化するような磁界が適しています．

以下，部分ごとに原理を述べます．

● 使用するコイル

本手法では2種類のコイルを使用します．空間に交流磁界を生成するためには，図1（a）に示す励磁コイルを使用します．励磁コイルは測定空間を囲む立方体状のコイルです（人間が中に入るためには一辺2m程度が必要）．立方体の6面おのおのが正方形のコイルになっています．使用する磁界は弱いので（CRTモニタがかすかに揺れる程度），20ターン程度と少ない巻き数です．3組の対向する面のコイルを対にして使用します．また，対向するコイルの中心を通る直線をコイル軸とし，計測座標系

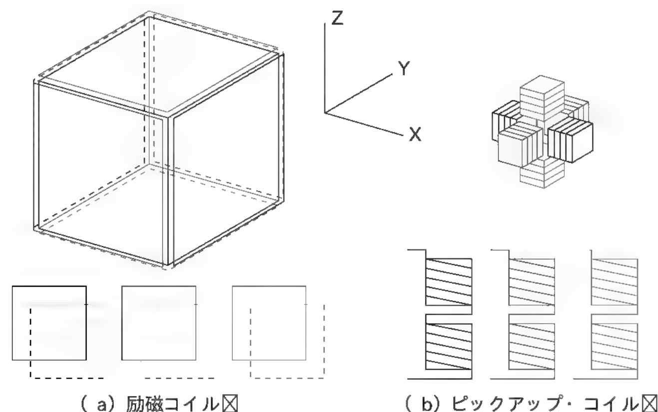


図1　モーション・キャプチャに使用するコイル

注1：映っているものに関する知識，前後関係の把握，物体の運動により2次元でもある程度は奥行きはわかる．

COLUMN

モーション・キャプチャ装置開発の動機

筆者が以前所属していた東北大学大学院工学研究科 江村研究室では，研究の一つとしてバーチャル・リアリティを用いたドライビング・シミュレータの開発が行われていました．ドライビング・シミュレータはゲームも含めて種々ありますが，このシステムは運転の体験に重きをおき，なるべく実車に近づけることを目的にしていました．これにより，日本の道路の運転練習のみならず，たとえば右側通行の国などでの運転感覚の事前体験による交通事故率の低下もねらいでした．そのため，開発にあたってはなるべくリアルに，さらに普及を考えてなるべく低コストになるようにと考えていました．

このシステムでは，コンピュータ上の仮想空間を車両が行き交います．人間はハンドルやペダル類，ミッションがある運転席(実車の一部)に座り，頭にはヘッド・マウント・ディスプレイ(HMD)と姿勢のセンサを取りつけます．すると，操作に従って自動車が走り，運転席からの光景が見えます．頭部のセンサにより，顔を向けた方向の風景が提示されるため，左右やミラー，メータ類を見ることができます．しかし，頭部の姿勢しか使用していないことが一つの問題でした．人間は運転時，頭の向きだけではなく，位置も動かします．たとえば，後方を確認するときは座席の間から後を振り返って見ますし，柱が邪魔ならば頭の位置をずらします．よりリアルなシステムにするには姿勢だけではなく，位置も必要になりました．

3次元での位置，姿勢を検出するモーション・キャプチャにはいくつかの方法があります．代表的なものに機械式，光学式，磁気式の三つがあり，その比較を表Aに示します．このほかに，ジャイロや加速度センサを用いる内界センサ式もありますが，位置の取得には向きません．

目的は人間の頭部動作をなるべく低コストに取得することであったため，磁気式の導入の検討を開始しました．すでにいくつかの手法が提案され，市販品もあったのですが，もともと開発系の研究室でもあり，研究担当者が既存手法の実験を始めたことをきっかけに，江村教授の発案でモーション・キャプチャそのものの開発研究もスタート(1)しました．

表A　モーション・キャプチャの方式による比較

方式	機械式(リンク式)	光学式(カメラ)	磁気式
原理	関節がある，腕状の装置(リンク)を対象に固定する．対象の動きに応じて関節が曲がり(伸び縮みし)，その角度(伸縮量)を計測し，演算することで対象の位置と姿勢を得る	対象に光点(小電球など)，反射のよい球などを取り付け，複数台のカメラで同時に撮影し，各カメラ映像のどこに対象が映ったかを演算することで位置を得る．姿勢は複数の点の相対位置から得る	測定対象空間に複数の磁界を発生させ，対象に取り付けたコイルでその磁界を検出し，演算によって位置，姿勢を得る．逆に対象で磁界を発生させ，周囲でそれを検出する方法もある
おもな利点	検出が確実，リンクが対象に追従すれば応答がよい	同時に多くの点を取得可能	比較的低コスト．磁界は人間を通るため，運動にともなう死角が発生しない
おもな欠点	機械的なものであるため，対象の運動を制約することが多い	人体の陰などのカメラの死角になると検出不能．1点の姿勢を求めにくい．非常に高コスト	磁性体が近くにあると磁界がひずみ，検出に誤差が生じる

のX，Y，Z軸とします．

対象には，図1(b)に示すピックアップ・コイルを取り付けます．小形で巻き数の多い空芯コイルを6個，直交する3軸の方向に取り付け，同軸上のコイルを直列に接続します注2．具体的には，小形のトランジスタ増幅回路用のトランスからコアを抜いて，アクリルで作った芯に固定しました．このコイルの軸が，求めるピックアップ・コイルの座標系になります．

● 交流磁界，コイル，誘導起電力の関係

コイルに交流電流を流すと，それにともなって交流磁界が生じます．私たちの身のまわりには，計測中は固定とみなせる地磁気や永久磁石が作り出す磁界や，さまざまな電子機器の信号に起因する磁界，商用電源の50/60Hzの磁界注3などが重畳して存在します．そのため，本手法では交流磁界を用い，同期検波を使用することで，周囲のその他の磁気の影響を低減しています．以下では，その前提のもとでその他の磁気を無視します．

次に，交流磁界とピックアップ・コイルの関係について図2を参考に簡単に確認します．コイルに誘導する起電力は$e(t)$は，コイルの巻き数n，コイルを貫く磁束$\Phi(t)$によって，

$$e = -n\frac{d\Phi}{dt} \quad \cdots\cdots (1)$$

と表されます．磁束の時間変化が電圧になります．この磁束はいわば磁力線の本数のようなもので，コイルで囲まれた面で，磁束密度$\boldsymbol{B}$(ベクトル場)を面積分して求めます．ただし，コイルの面内で磁束の方向も大きさも一定と見なせるならば，コイルの断面積をS，コイルの面の法線方向と磁束密度の方向がなす角度をθとして，

$$\Phi = S|\boldsymbol{B}|\cos\theta \quad \cdots\cdots (2)$$

で簡単に得られます($\theta=0$で最大，$\theta=\pi/2$，90°のとき0)．ピックアップ・コイルが十分に小さければこの仮定が成り立つ

注2：なるべく小さくて，一点で軸が交わるコイルが理想的だが，特注するとコストが高くつくので汎用品で作ってある．

注3：開発中にやたらと邪魔になる交流磁界があって，調べてみたら50Hzの高調波だった．PCなどが多く置かれた部屋だったことから，電源回路の平滑コンデンサに流れ込む瞬間的な電流の影響だと考えられる．

ので，以下ではこの式を用います．さらに，ピックアップ・コイルの方向ベクトル（コイル面の法線）を v（$|v|=1$）と置くと，ベクトルの内積を用いて，

$$\Phi = SB \cdot v \quad \cdots\cdots (3)$$

と書き換えられます．

● 姿勢の検出

姿勢を検出するための基本的な考え方は，「なるべく方向のそろった磁界を基準に，ピックアップ・コイルの方向を得る」というものです．この目的のため，対向するコイルを対として使い，ヘルムホルツ・コイルに似た磁界を生成します．

対向するコイルには，同じ方向に，同じ振幅，位相の交流電流を流します．この場合，図3（a）に示すように，両コイルで同じ方向に励磁し，励磁コイル内にほぼ平行な磁界が発生します．この磁界を協調磁界と呼んでいます．協調磁界のより細かな解析結果を図3（b）に示します．外枠の正方形が励磁コイル全体，外枠の左右の線が通電したコイルの面になります．励磁コイル内の各点での磁束密度（交流磁界の振幅）の方向と大きさを短かい線で表しました．なお，この結果は励磁コイルの中心を通る水平面での解析結果です．この結果から，コイルの巻き線の近辺（4隅）以外では大きく方向が変わらない磁界が得られることが確認されました．

そこで，この磁界中にピックアップ・コイルを置き，その誘起電圧信号から方向を得る方法を検討しました．まず，i 番目の対となる励磁コイルに同じ向きに角振動数 ω_i の電流を流します（i=1，2，3）．この磁界中にピックアップ・コイル j（j=1，2，3）を置き，その方向ベクトルを v_j とします．ピックアップ・コイルを貫く磁束は，その場の磁束密度を $B_{Ci} c_i \cos(\omega_i t)$ として，

$$\Phi_{i,j}(t) = SB_{Ci}\, c_i \cdot v_j \cos(\omega_i t) \quad \cdots\cdots (4)$$

によって得られます[注4]．

ここで，c_i は正規化された磁束密度場を表します．磁束密度そのものの強さは電流やコイルの大きさの影響を受けますが，そのコイル内での位置に依存した相対的な強さや方向については相似関係があります．そこで，励磁コイルの中心で大きさが1となるように調整したものを c_i としました．一方，B_{Ci} は電流やコイルの大きさなどの装置によって定まる定数です．これは装置を稼働させてから測定します[注5]．両者を乗じることで，磁束密度の振幅を得ます．

この磁束により生じる起電力は，ピックアップ・コイルの巻き数を n として，

$$\begin{aligned} e_{i,j}(t) &= -n\frac{d\Phi_{i,j}}{dt} \\ &= nS\omega_i B_{Ci}\, c_i \cdot v_j \sin(\omega_i t) \\ &= K_{Ci}\, c_i \cdot v_j \sin(\omega_i t) \quad \cdots\cdots (5) \end{aligned}$$

注4：i 番目の磁界と j 番目のコイルの関係であることに注意．
注5：実際には式（5）の定数 K_{Ci} を計測する．

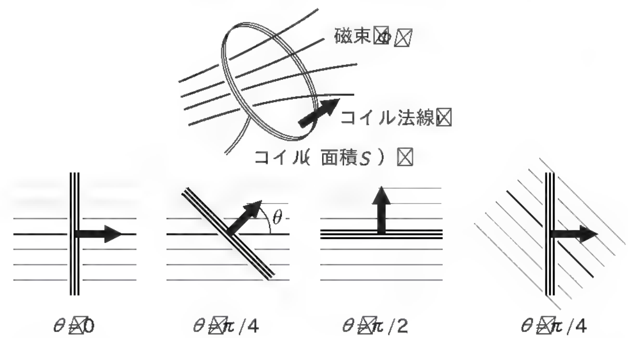


図2　磁界とコイルの関係

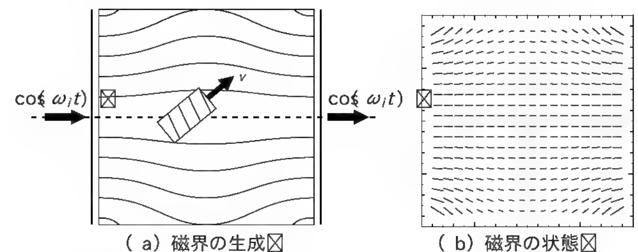


図3　協調磁界

となります．式の簡単化のため，定数部 $nS\omega_i B_{Ci}$ を K_{Ci} と置きました．

次に，誘起した電圧に参照波 $\sin(\omega_i t)$ を乗じて，角振動数 $2\omega_i$ を除去可能なローパス・フィルタを適用することで検波出力 $R_{i,j}$ を得ます．

$$\begin{aligned} r_{i,j}(t) &= e_{i,j}(t) \cdot \sin(\omega_i t) \\ &= K_{Ci}\, c_i \cdot v_j \{1 - 2\cos(2\omega_i t)\}/2 \quad \cdots\cdots (6) \\ R_{i,j} &= K_{Ci}\, c_i \cdot v_j / 2 \quad \cdots\cdots (7) \end{aligned}$$

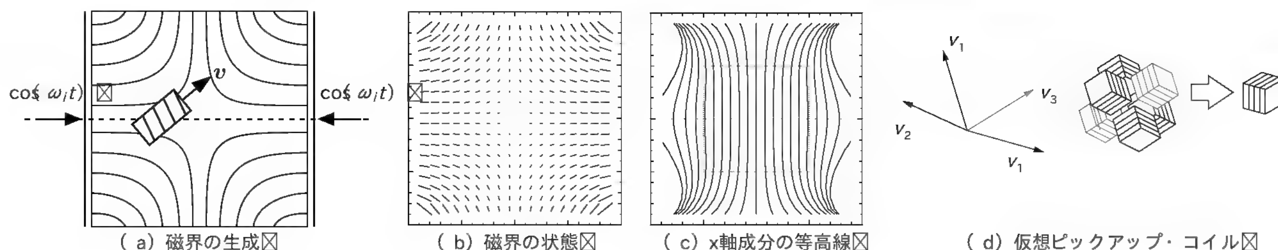
この検出値 $R_{i,j}$ が，各励磁コイルによる磁束密度のベクトル場とピックアップ・コイルの方向ベクトルの内積を含むことがポイントです．

さて，ピックアップ・コイル j について，求めるべき未知数はベクトルの3成分，すなわち v_{jx}，v_{jy}，v_{jz} です．一方，検波出力 $R_{i,j}$ は，励磁コイルが3対あるので3通り得られます．内積で表記した部分を内積の定義に従って書き直すと，

$$\begin{aligned} c_{1x} v_{jx} + c_{1y} v_{jy} + c_{1z} v_{jz} &= 2R_{1,j}/K_{C1} \\ c_{2x} v_{jx} + c_{2y} v_{jy} + c_{2z} v_{jz} &= 2R_{2,j}/K_{C2} \\ c_{3x} v_{jx} + c_{3y} v_{jy} + c_{3z} v_{jz} &= 2R_{3,j}/K_{C3} \quad \cdots\cdots (8) \end{aligned}$$

となります．ここで，$R_{i,j}$ は検出された値，K_{Ci} は装置定数なので右辺は既知，また $c_{ix/y/z}$ はあらかじめ数値計算で求めておくことができます．そのため，上式は単純な3元1次連立方程式になっていて，$v_{jx/y/z}$ を容易に求めることができます．ただし，$c_{ix/y/z}$ は位置がわからなければ値を確定することができません．そこで，磁界の方向がほぼ揃っていることを前提に，かりに励磁コイル中央の値を用いて仮の姿勢を求め，後に再計算します．

以上の計算を3個のピックアップ・コイルに対して行い，ピックアップ・コイルの座標軸を求めます．本来，この3ベク



(a) 磁界の生成　(b) 磁界の状態　(c) x軸成分の等高線　(d) 仮想ピックアップ・コイル

図4　差動磁界

トルは直交するはずですが，磁界の歪みなどの原因により直交せず，補正を要します．

● 位置の検出

姿勢に引き続き位置を検出します．基本的な考え方は，「位置に応じてなるべくリニアに変化する磁界をつくり，検出して位置を求める」です．最低限，何らかの単調増加的な変化があれば十分ですが，そのうえで比例関係があれば，装置のハードウェアが持つダイナミック・レンジを有効に使うことができます[注6]．

位置の検出には，図4に示すような磁界を用いました．対向する励磁コイルに同じ振幅の正弦波電流を逆向きに流し，逆向きの磁界を生成します．このとき，両励磁コイルから等距離にある面では，コイルに鉛直な成分は互いに打ち消しあって強度が0となり，それぞれの励磁コイルに近付くに従って逆位相で磁界の振幅が大きくなります．この磁界を差動磁界と呼んでいます．

差動磁界の数値解析結果を図4(b)に示します．各点での磁界密度の大きさ，方向は場所によって大きく異なり，そのままでは使いにくいように思えます．ところが，励磁コイル軸方向(コイル面に鉛直な方向)の成分のみに着目すると，図4(c)に示すような等高線が描けました．等高線がほぼ平行に並んでいるということは一定の傾斜を意味し，位置と磁界にリニアな関係があることが見い出されました．そこで，この軸成分を得ることを目標にします．

まず，姿勢の検出と同様に磁界を生成し，検出します．図4(a)に示したように，対向する励磁コイルに角振動数ω_i(i=4, 5, 6)で振幅が等しく逆向きの交流電流を流します．このとき，ピックアップ・コイルを貫く磁束および起電力は次式で与えられます．

$$\Phi_{i,\ j} = SB_{Di}\boldsymbol{d}_i \cdot \boldsymbol{v}_j\cos(\omega_i t) \quad \cdots\cdots (9)$$

$$e_{i,\ j} = nS\omega_i B_{Di}\boldsymbol{d}_i \cdot \boldsymbol{v}_j\sin(\omega_i t) \quad \cdots\cdots (10)$$

ただし，先ほど同様，差動磁界の磁束密度場を装置定数B_{Di}と正規化した磁束密度ベクトル場$\boldsymbol{d}_i$(励磁コイルの面の中央で大きさ1となるように正規化)の積$B_{Di}\boldsymbol{d}_i$として与えます．やはり同様に検波出力を得ます．

$$r_{i,\ j} = K_{Di}\boldsymbol{d}_i \cdot \boldsymbol{v}_j\sin(\omega_i t)\cdot \sin(\omega_i t)$$
$$= K_{Di}\boldsymbol{d}_i \cdot \boldsymbol{v}_j\{1-\cos(2\omega_i t)\}/2 \quad \cdots\cdots (11)$$

$$R_{i,\ j} = K_{Di}\boldsymbol{d}_i \cdot \boldsymbol{v}_j/2 \quad \cdots\cdots (12)$$

ただし，$K_{Di}=nS\omega_i B_{Di}$と置き換えました．この結果から，ピックアップ・コイルの方向ベクトル$\boldsymbol{v}_j$(姿勢の検出により既知)とベクトル場$\boldsymbol{d}_i$の内積が得られます．

次に，励磁コイルの軸成分を得ます．このためには，仮想ピックアップ・コイル法を用います[(1)]．

これは，図4(d)に示すように，方向ベクトル$\boldsymbol{v}_j$が既知である3個の直交する実在のピックアップ・コイルの出力強度から，任意のベクトル$\boldsymbol{v}_I$に沿い，実在するピックアップ・コイルの中心に位置する仮想的なピックアップ・コイルの強度を推定するものです．各ピックアップ・コイルで検出された振幅を$P_{i,\ j} = 2R_{i,\ j}/K_{Di}$とすると，仮想ピックアップ・コイルで得られる振幅$P_{i,\ I}$は次式で得られます．

$$P_{i,\ I} = \boldsymbol{v}_1 \cdot \boldsymbol{v}_I P_{i,1} + \boldsymbol{v}_2 \cdot \boldsymbol{v}_I P_{i,2} + \boldsymbol{v}_3 \cdot \boldsymbol{v}_I P_{i,3} \quad \cdots\cdots (13)$$

ただし，$|\boldsymbol{v}_I| = |\boldsymbol{v}_j| = 1$です．

このベクトル$\boldsymbol{v}_I$を，各励磁コイルの軸方向に設定して，図4(c)に示した強度分布と照合することで空間内でのピックアップ・コイルの位置が得られます．実際には，あらかじめ磁束密度場を計算することで，3軸方向の$P_{i,\ j}$から直接座標に変換できる数値テーブルを用いました．

● 姿勢と位置の計測

以上の処理法を組み合わせ，ピックアップ・コイルの位置と姿勢を得ます．

- ピックアップ・コイルの誘起電圧信号より検出値$R_{i,\ j}$(i=1, …, 6, j=1, 2, 3)を求める．
- ピックアップ・コイルの位置を励磁コイル中央と仮定し[注7]，仮の姿勢$\boldsymbol{v}_j$を求める．
- 得た姿勢をもとに位置を求める．
- 得た位置をもとに姿勢および位置を再計算する．

すでに述べたように，姿勢検出に必要な磁束密度場はピックアップ・コイルの位置が定まらなければ確定しません．そこで，大まかに姿勢を求めて，それによって位置を求めて，姿勢を求め直しています．

今回使用した協調磁界，差動磁界とも，位置に対する変化が極端ではないため，1回の再計算で十分な精度が得られました．

注6：一般には，磁界は距離の2乗分の1などに比例して減衰する．
注7：連続して検出を行う場合は，前回検出した位置を使うと，仮定による誤差が大幅に低減される．

● 原理のまとめ

以上のような原理を実装することで，3次元での姿勢と位置の計測が可能となります．

実装に必要なことは，

- 正弦波電流をコイルに流し，交流磁界を生成する
- 同期検波により振幅を求める
- 振幅から姿勢，位置を求める

となります．以下では，具体的にその実現方法および実装について述べます．

モーション・キャプチャの実装

以上の原理をもとに，実際に開発したプロトタイプについて紹介します．

● 信号の最適化

実装をなるべく容易にするために，原理をもとに実装用の信号の検討を行いました．

本手法で必要なのは，6種類の正弦波信号です．三角関数の性質(Appendix 参照，p.123)より，正弦波と余弦波は独立したものとして扱えます[注8]．そこで，3周波数の両信号を使います．

具体的には，1対の励磁コイルで協調磁界を生成するために $\cos(\omega_i t)$ を，差動磁界を生成するために $\sin(\omega_i t)$ を用います．複数の磁界を同時に生成する場合には，コイルを複数用意する必要はなく，同一のコイルに重畳した電流を流します．そのため，一方のコイルには，$\cos(\omega_i t)+\sin(\omega_i t)$ の電流を，もう一方には $\cos(\omega_i t)-\sin(\omega_i t)$ の電流を流すことになります．さらに変形すると，

$$\cos(\omega_i t)+\sin(\omega_i t)=\sqrt{2}\cos(\omega_i t-\pi/4)$$
$$\cos(\omega_i t)-\sin(\omega_i t)=\sqrt{2}\cos(\omega_i t+\pi/4) \quad \cdots\cdots\cdots\cdots (14)$$

となります．つまり，位相が±45°ずれた余弦波を使用するだけになります(図5)．単に余弦波をつくるだけなら，以下で説明するように簡単ですが，二つ以上の波形を加算するためにはその分の回路が余分に必要になるため，この簡略化は単純なようで効果的です．

次に使用する周波数の選定です．式(7)，式(12)で示したように，同期検波を行う際に，誘起信号と参照波を乗じて，ローパス・フィルタで不要周波数成分をカットする必要があります．原理式では明記していませんでしたが，実際には使用する周波数間でも相互に影響が出ます．具体的には，角周波数 ω_1 が誘起しているときに，参照波として ω_2 を乗じると，$\omega_1 \pm \omega_2$ の成分が発生するので，フィルタでカットする必要があります．問題になるのは低いほうの $\omega_1-\omega_2$ です．フィルタのカット・オフ周波数を十分に低くすれば除去はできますが，このフィルタは装置としての応答性も直接的に制限します．よって，応答性のためにはフィルタのカットオフをなるべく高くし，それで除去できるような周波数を使用することになります．もちろん，

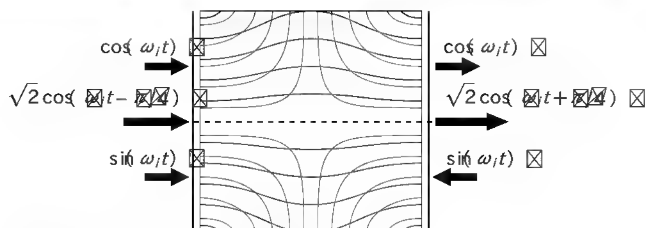


図5　協調磁界と差動磁界の同時生成

周波数を上げるほど回路設計への負担は大きくなります．

そこで，本装置では基本となる角周波数 ω_0 を定め，使用する励磁周波数は ω_0 の整数倍にすることにしました．この場合，任意の角周波数の組の差も ω_0 の整数倍になることが保証されるため，ω_0 を除去できるフィルタを設計すればよいことになります．おそらく，この選定法が周波数差を最大にしつつ，周波数の上限をもっとも低く抑えられる方法です．後述しますが，この選定法はディジタル信号処理を用いる際には，さらに利点を生みます．

最後に，整数倍の比率について検討します．単純には，「1，2，3倍」でもかまわないのですが，この場合，1倍の励磁信号に歪みがあり，高調波が発生した場合には，2，3倍の検出出力に混入してしまい，そもそも十分な分離が不可能になってしまいます．その観点からは，「互いにほかの倍数にならない」倍数の選定が効果的と考えられます．具体的には，3，4，5倍を使用しました．これにより，それぞれの高調波が発生しても互いに干渉することはなくなりました．

● ディジタル信号処理の導入

本手法は大部分をアナログ回路で実装可能です．一番最初に開発した試作機では，正確さ(同期)が要求される励磁信号，検波用参照波をカウンタとROM，D-A変換器で生成した以外は，アナログ乗算器やフィルタで構成しました．しかしながら，アナログ回路で精度や S/N 比を稼ぐことは容易ではないため，処理のディジタル化を進めました．本章で紹介する装置では，励磁信号の増幅部，誘起信号の増幅部以外はディジタル化し，大幅な性能向上を達成しました．

ディジタル化するうえで重視したことは，

- 有効桁数につねに配慮する．入力情報が桁落ちしないビット数を確保すると同時に，回路規模を抑えるために必要以上の桁数としない
- 「2の n 乗」を設計の前提にする．多少の半端には目をつむって 2^n にそろえる．これにより，タイミング生成のカウンタが簡素化し，乗除算がシフト演算＝線のつなぎ換えだけで済む

ということです．より大規模のロジックを使用すれば解決する問題もありますが，この方針では回路が単純化することもあり，

注8：直交性として知られている．通信分野でもよく使われている．たとえば，NTSC信号の色情報はサブキャリア周波数3.58MHzのcos/sinの振幅で伝送される．

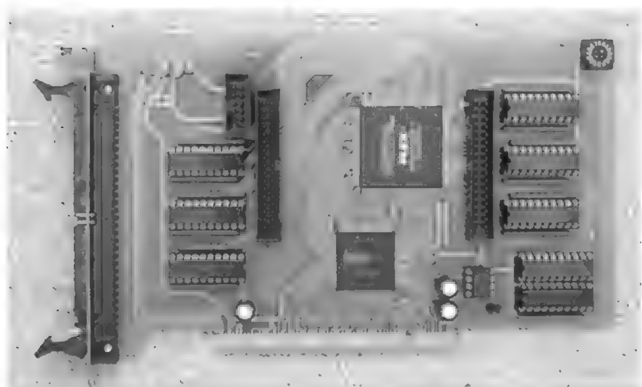
高速動作が容易となります．また，後述しますが，フィルタの設計もディジタルならではの手法を用いました．

● Universal Interface Board

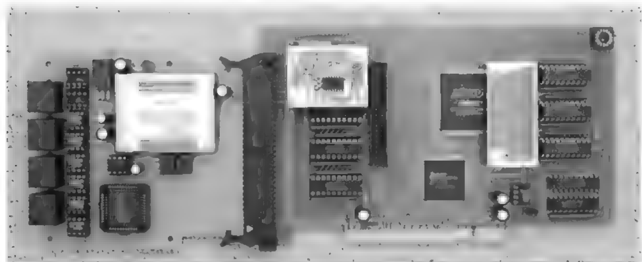
ディジタル回路の実装には，図6に示す，自作のUniversal Interface Board（UnivIF）[3]を使用しました．これはPCIバスの拡張ボードで，「何にでもなるディジタル・インターフェース・ボード」をめざして開発したものです．ボード上にはPCIバス信号をデコードして，昔ながらのアドレス，データ，制御バス信号に直してくれるアドテックシステムサイエンス社製のバス・ブリッジAPIC21と，ALTERA社製のPLDであるFLEX 6024を搭載しています．そのほかに，PLDのオンライン・コンフィグレーション用回路，外部信号の入出力バッファ，拡張コネクタなどを備えています．類似の製品が販売されていますが，このボードの特徴は以下のとおりです．

- PCIバス用のボードを自作することはスリルに満ちているが，物理的な配線は完成しているので，少ないリスクで特殊なインターフェースを作ることができる
- これ自体が完結したインターフェース・ボードであり，ほかに物理的な回路（バッファ，コネクタなど）をはんだ付けして加える必要がない
- PCが稼働中にも随時PLD上の回路を変更できるため，そのときの目的に応じた回路を1秒程度でダウンロードし，即稼働させることができる[注9]
- 使い終わったボードは別用途に再利用できる
- 安価である[注10]

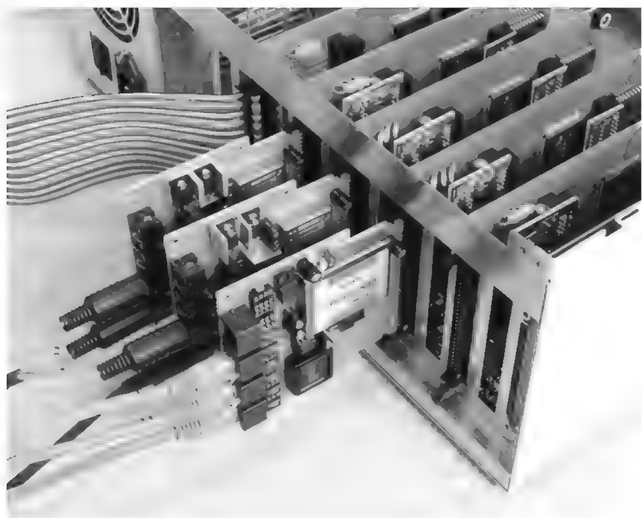
なお，本ボードの仕様はWebページ[3]ですべて公開しています[注11]．



（a）UnivIF本体



（b）アナログ波形出力用オプション装備（励磁信号発生回路）



（c）モーション・キャプチャ信号処理部

図6　Universal Interface Board（Univ IF）

● 励磁信号と参照波の生成

励磁信号および参照波の生成は同一の手法，Direct Digital Synthesizer（DDS）を用いました．励磁信号生成回路の概要を図7に示します．励磁信号の周波数，位相オフセット，振幅はPC側から指定します．

まず，周波数をもとに位相をつくります．処理のサンプリング周期ごとに，

$$R_P[k]=R_P[k-1]+R_F \quad \cdots\cdots(15)$$

と演算します．[k]は系列の番号（サンプリング周期ごとに増加）です．位相レジスタ$R_P[k]$は1周期ごとに周波数R_Fずつ増えます．これに位相オフセットR_Oを加え，余弦を求めます．余弦はUnivIF上に増結した32Kバイトの高速SRAM上に置いた数値テーブルと，ロジック回路を用いて得ました．

$$C[k]=\cos(R_P[k]+R_O) \quad \cdots\cdots(16)$$

これに振幅値R_Aを乗じて，D-Aコンバータより出力します．

各部の仕様としては，余弦関数回路の入力は正数16ビット（0〜2π相当），出力は整数16ビット（±32767），各指定数値R_F，R_O，R_Aおよび位相レジスタR_Pは正数16ビット，出力は13ビット（乗算器の上位13ビットのみ）としました．

たとえば，$R_F=1$の場合，65536サンプリング周期で出力の余弦波が1周期となります．$R_F=2$とすると，65536サンプリングで2周期となり，一般的には65536サンプリングでR_F周期が出力されます．つまり，R_Fに比例した周波数の余弦波が得られます．

注9：たとえば，本装置の場合，初期化中にPLLの動作チェック，RAMへのデータ・ダウンロードの回路を生成して目的を果たし，その後，処理用の回路をダウンロードする．

注10：人件費をゼロと計算するのが一番の原因だが…．

注11：製造・販売してくださる企業をお待ちしております．

この回路は励磁コイル1本のための回路で，6セットが必要です．1対の励磁コイルには±π/4だけ正確に位相をずらして同期した信号が必要なので，すべての位相レジスタを同期クリアする機能を別途用意しています．後述の同期検波用参照波と同じように正弦波と余弦波をいっしょに生成することもできます(Appendix参照)が，後に説明するキャリブレーションを行いやすいように，独立にしています．

● 励磁コイルとその駆動

ディジタル回路で生成した励磁信号はMaxim社製13ビット/8チャネルのD-Aコンバータ(DAC)MAX547でアナログ信号に変換し，コイルを電流駆動するためのアンプに入力します．

励磁コイルの駆動回路は励磁コイルのそばに置くため，D-Aコンバータからは離れます．励磁信号を引き延ばす際の最大の問題は信号間のクロストークです．クロストークにより励磁信号がほかのコイルに漏れると，そのまま精度低下につながります．そこで，Analog Devices社のSSM2142とSSM2143の差動伝送用ICを使用しました[注12]．

駆動回路を図8に示します．National Semiconductor社製のLM675パワーOPアンプ[注13]を用いた，定電流増幅回路になっています．SSM2143で受けた信号(終端10kΩ)はC_5，VR_1で低域を除去し，出力を粗調整します．R_4が電流検出抵抗で，R_5，R_6によりU_2が非反転増幅回路として動作します．なお，R_2，C_6，C_7は安定化用，R_3は出力保護です．ただし，現状では，接続する励磁コイルによっては若干発振気味です．

励磁コイルを電圧駆動するか，電流駆動するかという選択について検討しました．まず第一に，本来磁界は電流に比例して発生するため，電流駆動が妥当と考えられます．第二に定電流源は理想的にはインピーダンスが無限大です．交流磁界を発生させている中に，ショートさせたコイルを置いた場合には，そこでエネルギを吸収してしまうため，磁界に影響を与えます．駆動している励磁コイル以外の励磁コイルのインピーダンスが低い場合はこの状態になるため，電流駆動が無難と考えられました．

励磁コイルは，計測空間を完全に囲むことができる，(なるべく)非磁性体で立方体の枠に，ホルマル線などを巻いて作ります．これまで作ったコイルは，500mm，900mm，1900mmの3種類です．前の二つはホーム・センタで購入した樹脂のアングル12本をL字金具でねじ止め[注14]した枠です．最大のものは，30mm角のラワン材の角に10mm角の溝を堀り，L字金具などで接合しました[注15]．コイルは直接巻いたのではなく，あらかじめちょうどよい寸法の正方形のコイルを作り，枠を組み立てた後ではめ込んでいます[注16]．

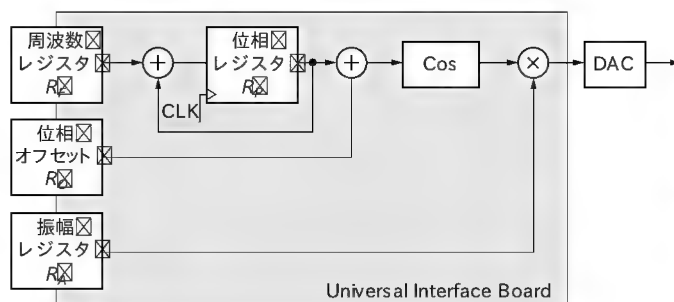


図7 励磁信号生成回路のブロック図

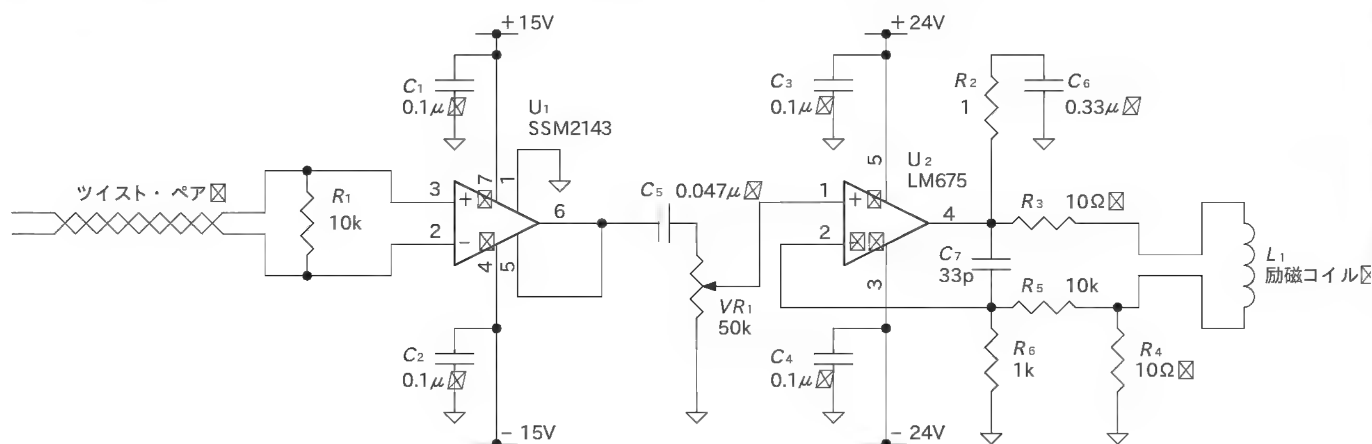


図8 励磁コイルの駆動回路

注12：接続には手軽に入手できるツイスト・ペア・ケーブルとして，100Base-TX用のCAT5ケーブルをそのまま流用した(コネクタがRJ45)．信号自体はそれほど高速ではないため，インピーダンス・マッチングはとっていない．市販の規格ケーブルを流用するように設計すると，ケーブル工作の手間が減り，コスト自体下がる．突発的に長さを変えたくなったときにも楽．

注13：±30V，3Aとそこそこ大きめの手軽なパワーOPアンプ．東京・秋葉原の秋月電子通商でも入手可能．

注14：本来は金具は使いたくないところだが，強度や組み立て性から使用した．この程度なら影響は少ないようである．

注15：夏に研究室の学生さんたちと木くずだらけになりながら作った．

注16：床の4点にフック付きの吸盤を吸着させ，ホルマル線のドラムを持ってぐるぐると回った．その後，ホットメルトを適宜塗布して，はんだごてを改造して作った加熱しごき器でホルマル線を固めた．

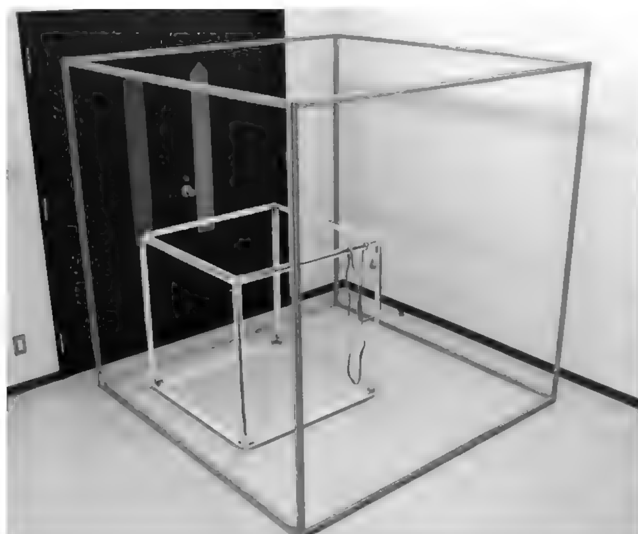


図9　励磁コイルの外観（1900mm × 1900mm × 1900mm）

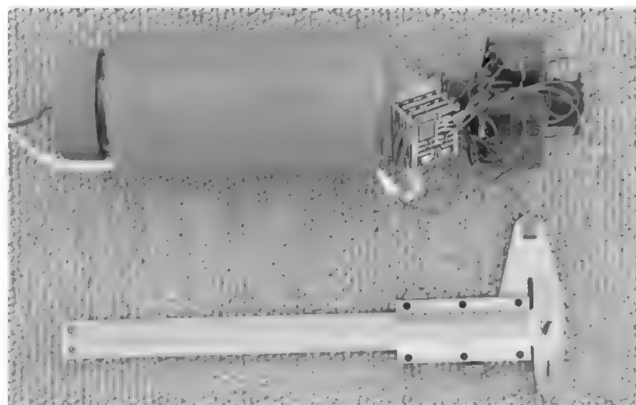


図10　ピックアップ・コイルの外観（コイル本体は左先端部のみで，左は入力用のON/OFFスイッチを兼ねたグリップ）

現在使用している図9に示す1900［mm］仕様のコイルでは，0.5［mm］のホルマル線を20回巻いています．直流抵抗は14Ω程度でした．

● ピックアップ・コイルと誘起信号

ピックアップ・コイルを図10に示します．ピックアップ・コイルは橋本電気社製の山水トランスST-14からEI鉄心を抜き，さらに入出力の両コイルを直列接続にしたものを6個を，アクリル樹脂製の治具で組み立てたものを使用しています．ST-14を選定した理由は，その小ささとインピーダンスの高さです．ピックアップ・コイルは巻き数が多いほど高感度になることは明らかで，同じ大きさならばインピーダンスや直流抵抗が高いトランスほど巻き数は多いはず，という選定を行いました．実数は不明ですが，単純なコイルと仮定して，コイルの径やインダクタンスから約8,000巻き程度と推定しています．

ピックアップ・コイルの出力は小振幅であり，ハイ・インピーダンスで受ける必要があります．また，ピックアップ・コイルの周囲に大きな回路を置くことは使い勝手を悪化させます．そこで，ピックアップ・コイルの直近の回路で1段目の増幅を行い，多少離れた2段目の増幅および伝送回路に接続することにしました．図11に回路図を示します．インスツルメンテーション・アンプの半分のような部分の1段目がピックアップ・コイルに隣接した回路です．ここから，極細のツイスト・ペア・ケーブル[注17]で2段目の同型アンプに信号を伝え，増幅して，励磁信号にも用いたSSM2142/2143ペアで伝送しました．

なお，1，2段目を接続するケーブルが4対であったため，増幅の基準点は正負電源を抵抗で分圧して作りました．また，適宜コンデンサを並直列に加えることで，全体でバンド・パス特性を持たせてあります．

以前の試作機ではこういった配慮をほとんど行っておらず，単に増幅してシングルエンドで接続していました．そのため，耐ノイズ性が悪く，クロストークもありました．今回のような配慮を加えたことで大幅に性能が向上し，その他の改良とあわせて，一桁以上の分解能の向上を果たし，処理後の検波出力のS/N比で約80dBを達成しています．

差動伝送した信号はSSM2143で受信し，Maxim社製の14ビット308kspsのA-DコンバータであるMAX121でディジタル値に変換し，受信処理用UnivIFに入力します[注18]．

● 同期検波のディジタル実装

誘起信号から励磁成分の振幅を取り出す同期検波は，式（6）および式（7）で説明しました．これをディジタル回路で実装します．処理回路のブロック図を図12に示します．この図は1コイルの誘起信号の1励磁周波数用なので，ピックアップ・コイル1個につき3セット，全体で9セット必要です．

参照波を作る方法は励磁信号と同等です．

$$R_P[k]=R_P[k-1]+R_F$$
$$C[k]=\cos(R_P[k]+R_O) \quad \cdots\cdots (17)$$
$$S[k]=\sin(R_P[k]+R_O)$$

ただし，振幅調整は不要であり，かつ，協調磁界，差動磁界用に，cos/sinの2波を作り出します．これをA-D変換値と乗じて，ディジタル・フィルタにかけます．

$$R_C=F(C[k]A_{in}[k]) \quad \cdots\cdots (18)$$
$$R_S=F(S[k]A_{in}[k])$$

出力値はPCIバス経由でソフトウェアが読み取り，演算を行います．

● ローパス・フィルタの選定と実装

すでに述べたように，同期検波用のフィルタが，本装置の性能を大きく左右します．励磁周波数やその他ノイズを除去できなければS/N比が悪化し，通過域が狭いと検出の応答速度が制限されます．

注17：断面積が4×1mm程度の極細のCAT5ケーブルが販売されていて，これを切って使っている．被覆を剥くのに難儀するが，とてもしなやかで，おすすめ．
注18：UnivIF内にMAX121の制御回路を生成した．

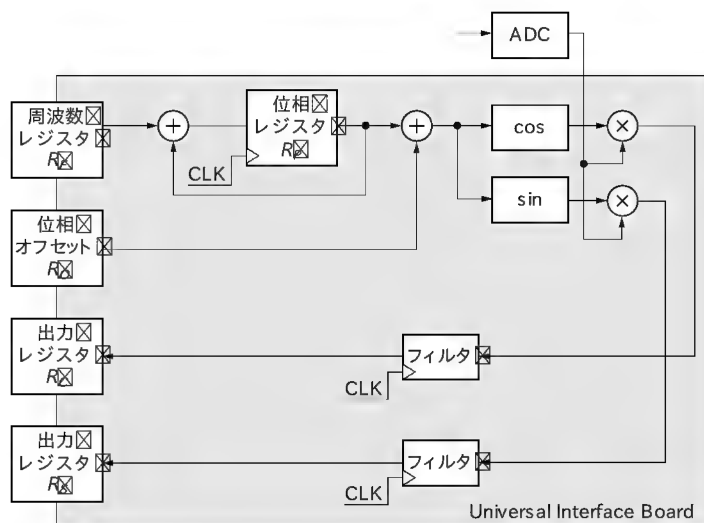


図12　同期検波回路のブロック図

そこで，一般的なノイズ除去用に1次のIIR型ディジタル・フィルタを，キャリア除去に区間平均型フィルタを使用しました[注19].

▶1次IIRフィルタの実装

IIRフィルタは，ディジタル・フィルタでよく解説されるFIR型[4]とは異なり，少ない処理でアナログ・フィルタと似た特性を得やすいフィルタです．1次の場合は，たとえば出力$y[k]$を入力系列$u[k]$と前回の出力$y[k-1]$で次式のように求めます．

$$y[k]=(1-r)y[k-1]+ru[k] \quad \cdots\cdots (19)$$

ここで，rは遮断特性を決める係数で，r=1の場合は入力が出力にそのまま通過します[注20].

ここで，$r=1/2^n$とおくと，以下のように変形できます．

$$y[k]=(1-1/2^n)y[k-1]+(1/2^n)u[k]$$
$$=(1/2^n)\{(2^n-1)y[k-1]+u[k]\} \quad \cdots\cdots (20)$$
$$2^n y[k]=\{2^n y[k-1]+u[k]-(1/2^n)2^n y[k-1]\}$$

この式をもとに回路ブロック図にしたものを図13に示します．仕様は入出力16ビット，n=10です．ディジタル・フィルタでは過去の値を保持するレジスタが不可欠ですが，ここでは$2^n y[k]$を保持します(26ビット)．演算式の$1/2^n$の部分は，10ビット分シフトして，レジスタの下位10ビットを切り捨てることにします．

このように，係数rを$1/2^n$と置いたことにより，加減算回路のみで実装できるようになります．また，ソフトウェア実装ではシフト演算は必要になりますが，ハードウェアでは配線をず

注19：アナログ試作機のときは2次のローパス，ディジタル化1号機は1次IIRローパスを3段使用していたが，いずれも帯域を大幅に制限していた．

注20：本格的な信号処理に限らず，ノイズっぽいデータをならすときなどに便利．rは直感もしくは数値計算で決める．また，処理手順によっては$u[k]$の代わりに$u[k-1]$を使用する場合もある．

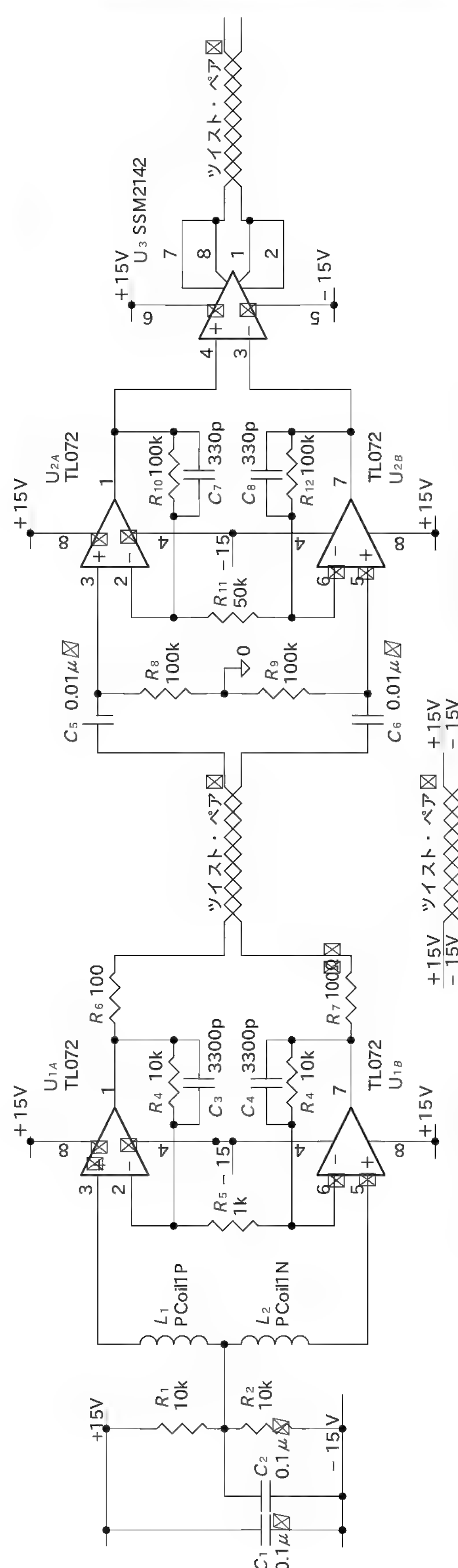


図11　ピックアップ・コイル・アンプ

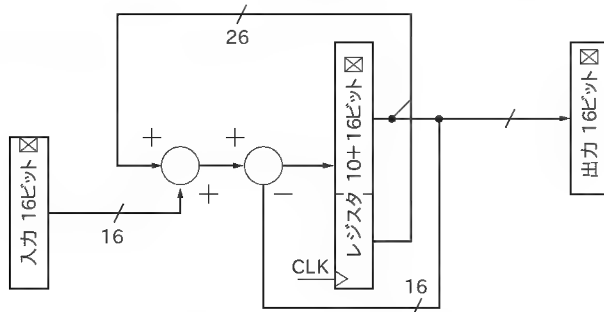


図13　1次ディジタル・ローパス・フィルタの実装

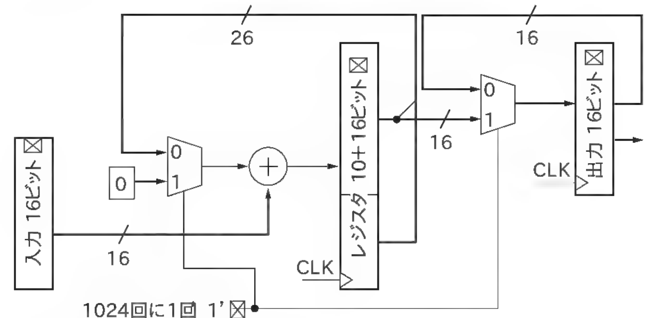


図14　区間平均フィルタの実装

らすだけで実装でき，お手軽です．

▶ 区間平均フィルタの実装

IIRフィルタは少ない次数で低めのカットオフ周波数を得やすいフィルタですが，どうしても遮断特性が緩くなってしまいます．本装置の場合，励磁周波数およびその差の成分をいかに除去するかが重要であり，以下に述べる区間平均フィルタを採用しました．

フィルタの出力$y[k]$は入力系列$u[k]$により次式で得ます．

$$y[k]=(1/N)\sum_{i=0}^{N-1}u[k-i] \cdots\cdots (21)$$

これはもっともポピュラなFIRフィルタであるN点移動平均フィルタです．式は同じですが，実装が異なるため，区間平均と呼んで区別しました．移動平均では，任意のkに対して，つねに上式が成り立つように演算します．そのため，$u[k-i]$の保持にはN個の保管レジスタが必要になります．それに対して，区間平均では，この演算をN入力につき1回しか行いません．次の$N-1$入力があるまで，出力は一定値のままです．

別の見方をすれば，移動平均フィルタの出力をN回に1回の間隔でホールドしたものともいえます．そのため，厳密には移動平均と特性は異なりますが，平均周期より長い間隔で結果を参照する場合は同等の特性を持つと考えられます．

具体的な実装を図14に示します．シフト演算を使用可能とするため，$N=2^n$とし，ここでは，$n=10$，$N=1{,}024$としました．通常はセレクタで入力'0'が選択され，演算用のレジスタは入力を順次加算し，出力レジスタは現在値を保持します．1024回に1回の割合でセレクタが'1'に切り替わり，演算レジスタは0にクリアされ，出力レジスタは演算レジスタの上位16ビット，すなわち演算結果を格納します[注21]．移動平均では入力をN個保持する必要がありますが，区間平均では現在値と結果の保持のみになります．

注21：出力レジスタについては，セレクタ＋Dフリップフロップに代えて，イネーブル付きDフリップフロップ(ALTERA社の設計シンボルではDFFE)を使うこともでき，見た目がシンプルになる．
注22：A-Dコンバータの変換速度が現在の制約条件．
注23：周波数レジスタR_Fは64の倍数とする．

このフィルタはディジタル化にともなう性能向上に大きく寄与しました．その特性について具体的な数値を交えて解説します．同期検波回路において，誘起した励磁信号($\omega_p=n\omega_0$)と参照波($\omega_r=m\omega_0$)の任意の組み合わせが発生します(n，mは整数)．このとき，三角関数の積和公式により，

$$\begin{aligned}&\sin(\omega_p t+\theta_p)\sin(\omega_r t)\\&=\sin(n\omega_0 t+\theta_p)\sin(m\omega_0 t)\\&=-(1/2)[\cos\{(n+m)\omega_0+\theta_p\}\\&\qquad-\cos\{(n-m)\omega_0+\theta_p\}] \cdots\cdots (22)\end{aligned}$$

となります．ここで必要とするものは，後者で$n=m$，すなわち励磁周波数と参照波周波数が一致したもののみで，$(n+m)\omega_0$と$(n-m)\omega_0$，$(n\neq m)$はフィルタで除去しなければならない成分です．この除去すべき成分はすべてω_0の整数倍になっています．一方，三角関数の性質として，整数$\ell\neq 0$に対して，

$$\sum_{i=0}^{N-1}\cos(2\pi\ell i/N)=0,\ \sum_{i=0}^{N-1}\sin(2\pi\ell i/N)=0 \cdots\cdots (23)$$

$$\sum_{i=0}^{N-1}\cos(2\pi\ell i/N+\theta)=0 \cdots\cdots (24)$$

が成り立ちます．

さて，本装置では，すべての動作の基準となる(メイン)クロックを66.6MHz(ホストPCのPCIクロック×2)とし，全チャネルの処理を256クロック[注22]で行っています．これが処理のサンプリング周波数になり，約260kHzです．さらに，1024分周した254Hzを波形生成および処理の基本周波数$f_0=\omega_0/2\pi$としています[注23]．前述のように，この3，4，5倍の周波数を励磁に使用しています．この1024サンプリングはまた，区間平均フィルタの平均区間にもなっています．励磁信号は，前述の回路によって，

$$E[k]=\cos\left(\frac{2\pi nk}{1024}+\theta_p\right) \cdots\cdots (25)$$

という形で生成されます．参照波も同様です．その結果，乗じた結果には，

$$p[k]=\cos\left\{\frac{2\pi(n\pm m)k}{1024}+\theta_p\right\} \cdots\cdots (26)$$

が含まれます．ここで，$n\pm m=0$の場合の平均値は，

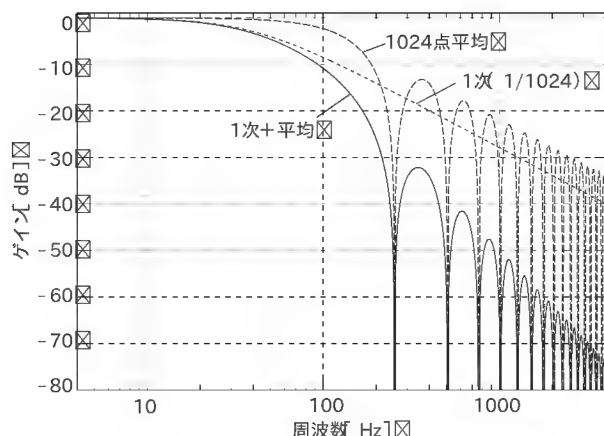


図15　1次IIRフィルタと区間平均フィルタの特性

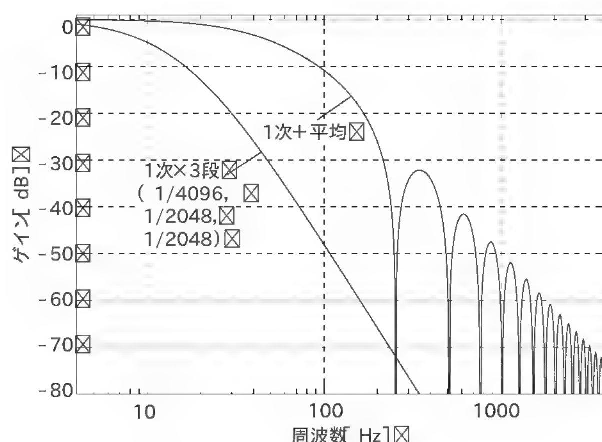


図16　IIRフィルタのみと区間平均フィルタ併用の特性の比較

$$\frac{1}{1024}\sum_{i=0}^{1023}\cos\left\{\frac{2\pi(n\pm m)i}{1024}+\theta_p\right\}=\cos(\theta_p) \quad \cdots\cdots\cdots\cdots (27)$$

となり，それ以外の場合は，

$$\frac{1}{1024}\sum_{i=0}^{1023}\cos\left\{\frac{2\pi(n\pm m)i}{1024}+\theta_p\right\}=0 \quad \cdots\cdots\cdots\cdots (28)$$

となります．つまり，乗算によって生じる不要な成分を，区間平均は確実に除去できるわけです．

非常に単純な回路で実現できる手法でありながら，“ディジタルであること”を有効活用して，キャリア成分をカットするという目的を達成することができました．

▶フィルタの伝達特性

最後に，フィルタの伝達特性を示します．図15に，1024点区間平均(移動平均)，1次IIRローパス・フィルタ(r=1/1024)，およびその合成特性を示します．サンプリング周波数は，本システムの実数値である，260kHzを想定しました．

区間平均(移動平均)フィルタでは，区間に一致する周波数(254Hz，508Hz，…)のところで急峻に除去されています．それに加えて，ローパス特性ももっています．現実には，キャリア以外にも多くのノイズを拾っており，その除去には単純に1次のローパス・フィルタを加えて補っています．

図16に，今回採用したフィルタと，以前のシステムで使用していたフィルタの特性を示します．以前は，1次IIRフィルタ(r=1/4096，2048など)を3段に接続していました(サンプリング周波数は同一)．除去しなければならない周波数で最低の254Hzを十分にカットするため，3段(3次)にしてかつカットオフ周波数をかなり下げていました．それにより応答も悪いものとなっていました．今回は，区間平均の採用で通過域を広くとることができ，応答性が大幅に向上しました．

● パイプライン化によるハードウェアの高密度と高速化

これらの処理は少ない素子で回路化しやすいように検討しましたが，数が多いため，単に並べると規模が大きくなってしまいます．そのため，処理回路を時分割で共用するようにして規模を抑え，さらにパイプライン化によって高速化しました．

パイプライン化対象の回路は2種類に分けられます．一つは値の保持を必要としない回路で乗算器や余弦波参照回路です．これらは回路にDフリップフロップを適宜挟み(もしくは回路プリミティブをパイプライン設定にする)，タイミングを検討するだけで済みます．

もう一つは値の保持が必要な参照波生成回路(位相レジスタ)，フィルタ回路です．これらの回路は，単純にデータを順に流すと保持用のレジスタで値が混合され，用をなしません．そこで図17に示すような回路構成を検討しました．第1案の図(a)は回路ごと並列化し，セレクタで切り替える方法ですが，そもそも時分割ではありません．第2案の図(b)は保持用のレジスタのみをセレクタで切り替える方法です[注24]．演算回路を時分割で使用するため，回路規模は図(a)より小さくなります．

もう一段検討したのが図(c)です．図(b)ではセレクタでレジスタを切り替えますが，入力データの順が毎回同じならば，切り替える順序はつねに一定です．そこで，セレクタを捨て，単なるシフトレジスタをつなぐことにしました．

動作を図18で説明します．図ではレジスタが3本あります．たとえば，フィルタの場合，ある時点で系列Aの入力データとレジスタの出力aをもとに次のレジスタ値を計算します．これは次のクロックで1個目のレジスタにラッチされます．その後，2個の別の入力B，Cがあり，クロックも2個入ると，aは3段目のレジスタまでシフトされます．そこで次の系列Aデータがくると，次のAの処理を行うことができます．

このような実装方法は既存のシステムで使われていると思われますが，参考書などで見あたらないため，RRF(Revolving Register File)と呼んでいます[注25]．処理数を増やすには，その数だけシフトレジスタを置けばよいため，動作速度はほとんど変わらず，FPGAなどで実装する場合も単純にDFFを消費す

注24：一般のCPUはある意味このような構造である．

注25：原案を書いた時点で，リボルバが思い浮かんだため，最初の呼び名は“Revolver”RFだった．

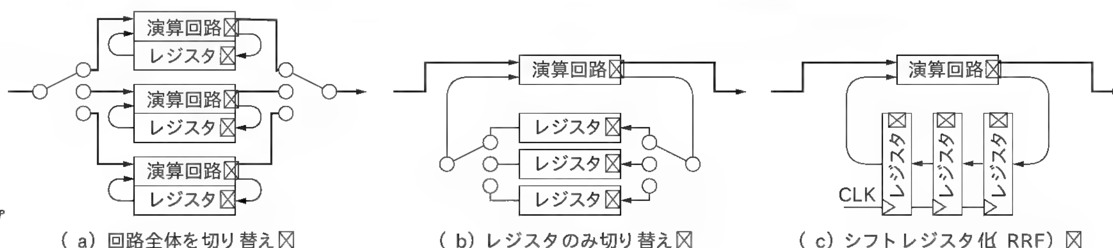


図17 演算回路のパイプライン化

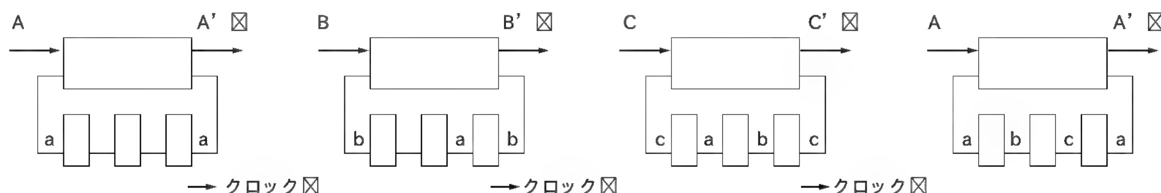


図18 RRFの動作概要

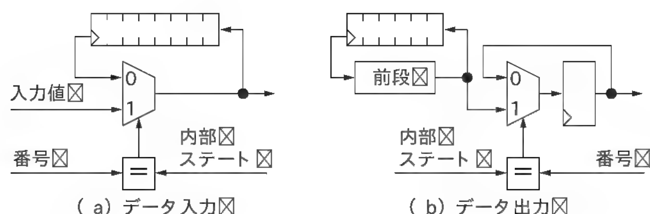


図19 PCとの時分割データ入出力回路

るだけです．それに対して，セレクタ型の場合はセレクタの回路規模が大きくなり遅延も大きくなるなど，実装上の問題が出やすくなります．

この回路方式は，図19に示す形でPC側との値のやり取りにも使用しています．PC側からデータ番号と値を指定すると，内部の現在処理中のデータ番号と一致したときにセレクタが切り替わり，入力用RRFに取り込まれます．一方，出力時は番号を指定すると，その結果が出力ラッチにコピーされます．入出力レジスタを多数並べてセレクタを使う方法に比較して回路は圧縮されますが，1回のデータのやり取りに，最低でもRRFの1サイクル分の時間を要することが弱点です．

以上の概念を元に，信号処理部分を実装しました．図20にピックアップ・コイル2個分のタイミング・チャートを示します[注26]．外部RAMへのアクセスを含む余弦波参照回路が4クロック消費するため，4クロックを一つの処理単位(メジャー・ステート)としました．参照波生成部は，余弦・正弦を求めるまでは周波数，位相が共通で，その後独立した処理になります．なお，処理によって，現在の処理データをそのまま次段に流す部分と，1サンプリング後にデータを流す部分があります．

● 姿勢と位置の検出

姿勢と位置の検出は，すでに述べた検出原理をもとに行います．プログラムのうち，中枢部をリスト 1に示します．プログラム中，関数`LoopupTableS`は与えられた位置における，コイル軸がX軸である励磁コイル対が作り出す協調磁界のベクトルを，事前に作成しておいた数値テーブルを補間して返す関数です．現在はコイルが立方体で対称であるため，使い回して3軸分求めています．また，`LookupPosition`は与えられた三つの仮想ピックアップ・コイル出力から，同様に位置を返す関数です．

● 磁束密度場の計算

本手法では磁界のようすがあらかじめわかっていることが前提になっています．磁界は数値計算で求めました．数値計算には電流と磁界の基本法則である，ビオ・サバールの法則を使用しました．「トランジスタ技術」，2004年8月号の別冊付録[5]にも解説があります[注27]．

$$d\boldsymbol{B} = \frac{\mu}{4\pi}\frac{I d_S \times \boldsymbol{r}}{|\boldsymbol{r}|^3} \cdots\cdots (29)$$

この法則は，電流Iが流れている経路のごく一部dsが，そこから$\boldsymbol{r}$離れた点に作る磁束密度を得る式です．μは透磁率で今回は空気中なので，真空の透磁率$\mu_0=4\pi\times 10^{-7}$を使用します．電流がある地点に作り出す磁束密度を求めるには，

- 電流経路を細かく刻む
- 刻んだ部分(電流素片)に相当するベクトル(ds)を求める(終点座標− 始点座標)
- 電流素片から，計算地点までのベクトル($\boldsymbol{r}$)を求める(計算地点− 電流素片の中点)
- 上式により磁束密度$d\boldsymbol{B}$を求め，すべての電流素片がつくる磁束密度を積算する

という手順を踏むことになります．ある程度までは刻めば刻むほど精度は良くなりますが，dsが$\boldsymbol{r}$より十分小さくなれば，それ以上刻む必要はありません(案外粗くとも大丈夫)．

注26:「トランジスタ技術」誌で以前紹介した「タイミングチャート清書ツール tchart」[6]は，まさに本装置のタイミング設計を行うために開発したもの．
注27: この付録では，磁束の単位が[Wb]ではなく[本]になっているなど，一般的な電磁気の教科書と多少異なるが，電気と磁気の関係の入門としておすすめ．

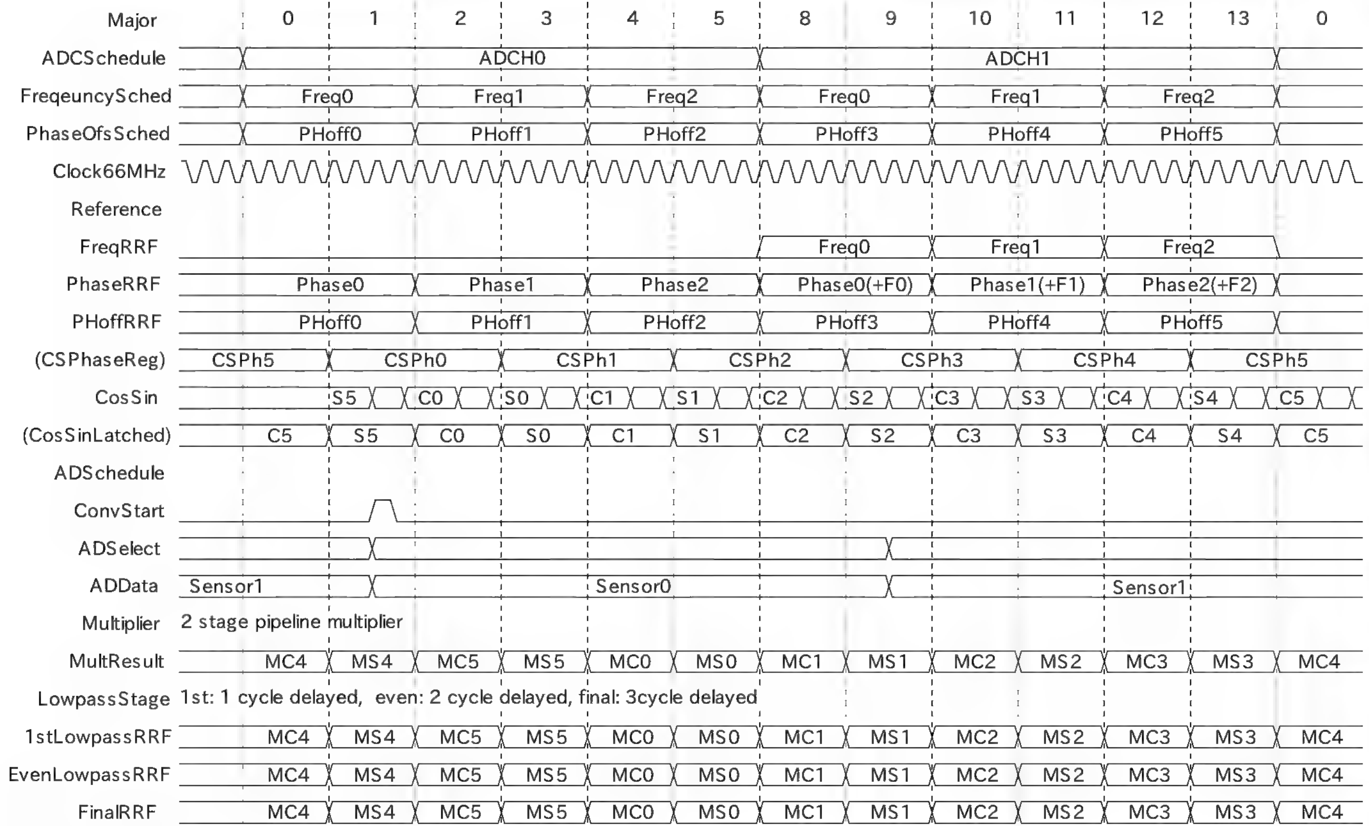


図 20　同期検波回路のタイミング・チャート

リスト 1　演算処理の中心部のプログラム

```
void GetMagcap(V3 &rpos,M3 &ratt)
{
  double p[3];        // 現在の座標
  double cop[3][3];   // 協調場検出値[pick][axis]
  double dif[3][3];   // 差動場検出値[pick][axis]
  double cfd[3][3];   // 協調ベクトル場
  double vct[3][3];   // ピックアップ・ベクトル[pick]
  int sel,ch;

  p[0]=p[1]=p[2]=0;    // 中心を仮定
  // データ取得(振幅校正済み)
  for(sel=0;sel<3;sel++)
    for(ch=0;ch<3;ch++)
    {
      cop[sel][ch]=RPReadAmplitudeReg(sel,ch,0);
      dif[sel][ch]=RPReadAmplitudeReg(sel,ch,1);
    }
  }
  // 1回目=======
  // 協調場ベクトル場行列　 LookupTableS は
  // X軸方向の場を返す= xyzを rotateして算出
  LookupTableS(p[0],p[1],p[2],
               cfd[0][0],cfd[1][0],cfd[2][0]);
  LookupTableS(p[1],p[2],p[0],
               cfd[1][1],cfd[2][1],cfd[0][1]);
  LookupTableS(p[2],p[0],p[1],
               cfd[2][2],cfd[0][2],cfd[1][2]);
  // 方程式解
  SolveEquation13(cfd,cop[0],  vct[0]);
  SolveEquation13(cfd,cop[1],  vct[1]);
  SolveEquation13(cfd,cop[2],  vct[2]);
  Normalize(vct[0]);  Normalize(vct[1]);
  Normalize(vct[2]);  // 大きさを1に修正
  // 直交性確保: ベクトルを強制的に直交させる
  OrthonormalizeVector(vct[0],vct[1],vct[2]);
  // 位置計算
  CalculatePosition(vct[0],vct[1],vct[2],dif, p);

  // 2回目=======
     (同上)

  for(int i=0;i<3;i++)
  {
    ratt[i][0]=vct[i][0];
    ratt[i][1]=vct[i][1];
    ratt[i][2]=vct[i][2];
    rpos[i]=p[i];
  }
}

void CalculatePosition(V3 &vx, V3 &vy, V3 &vz,
                       M3 &diff, V3 &p)
{
  double m[3]; // 仮想ピックアップ・コイル出力

  // x IPC ((1,0,0)*(vx[i],vy[i],vz[i]))*diff[i][0]
  m[0]=vx[0]*diff[0][0]+vx[1]*diff[1][0]+
       vx[2]*diff[2][0];
  // y IPC ((0,1,0)*(vx[i],vy[i],vz[i]))*diff[i][1]
  m[1]=vy[0]*diff[0][1]+vy[1]*diff[1][1]+
       vy[2]*diff[2][1];
  // z IPC ((0,0,1)*(vx[i],vy[i],vz[i]))*diff[i][2]
  m[2]=vz[0]*diff[0][2]+vz[1]*diff[1][2]+
       vz[2]*diff[2][2];

  LookupPosition(m[0],m[1],m[2],p[0],p[1],p[2]);
}
```

リスト 2　磁束密度の計算（誌面の都合上，文法を一部崩した）

```
// 真空透磁率μ0
const double mu0=4*M_PI*1e-7;  // [N/A^2]
// 距離分割単位長さ
const double LEN_DELTA=0.05;   // [m]
// シミュレーション実行部

// 線素の磁場（ビオ・サバールの法則）
void Calculate_DS(double rx,ry,rz, //位置(r)
        double dx,dy,dz,            // 線素ベクトル(ds)
        double I,                   // 電流
        double &bx,&by,&bz)         // 磁束密度(B)
{
  // dB = μ0 I ds × r / (4π|r|^3)
  double r=sqrt(rx*rx+ry*ry+rz*rz);
  double c=I*mu0/(4*M_PI)/(r*r*r);
  // B += dB = c*ds × r
  bx+= c*(dy*rz-dz*ry);
  by+= c*(dz*rx-dx*rz);
  bz+= c*(dx*ry-dy*rx);
}

// 線電流の磁場
void CalculateLineCurrent(
        double x,y,z,       // 観測点
        double sx,sy,sz,    // 始点
        double ex,ey,ez,    // 終点
        double I,           // 電流
        double &bx,&by,&bz) // 磁束密度
{
  int j,li;
  double l;                 // 長さ
  l=hypot(sx-ex,hypot(sy-ey,sz-ez));
  li=(int)(l/LEN_DELTA)+1;  // 分割数
  for(j=0;j<li;j++)
  {
    // 分割した線素の中点の内分率
    double r=(j+0.5)/li;
    // ビオ・サバールの計算
    Calculate_DS(
          x-(sx*(1-r)+ex*r), // r=
          y-(sy*(1-r)+ey*r), //線素中点
          z-(sz*(1-r)+ez*r), //〜観測点
          (ex-sx)/li,(ey-sy)/li,
          (ez-sz)/li,        // 線素
          I,                 // 電流
          bx,by,bz);         // 磁束密度(戻)
  }
}
```

以上をプログラムにしたものをリスト 2に示します．これは実際に磁場計算を行うために使っているプログラムの一部です．関数 `Calculate_DS()`がビオ・サバールの法則を演算している部分で，`CalculateLineCurrent()`は`Calculate_DS()`を呼び出しつつ，線分に流れる電流による磁界を計算します．これを 4辺集めると励磁コイルが作る磁界になります．この計算を空間にわたって行いました．

● キャリブレーション

最後に，キャリブレーション（校正）について触れておきます．本装置は調整すべき項目が 2点あります．

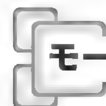
1点目は位相です．回路規模的に，UnivIF1枚にはすべては収まらないため，クロックは同期していても[注28]，励磁・同期検波回路間で回路内のカウンタ類が一致していません．また，各種回路を通る間にも位相ずれは発生するため，励磁信号と参照波の同期をとる必要があります．

そのため，まず，ハードウェアの初期化終了後に協調磁界だけを発生させ，ピックアップ・コイルでそれを検出して位相あわせを行います．この際，励磁信号発生側の位相オフセット・レジスタをゼロ[注29]にして，位相の調整は同期検波側のオフセット・レジスタで調整します．

次に振幅の校正を行います．励磁コイル立方体の内部で，振幅の測定に適している箇所は，コイル中央部です．ここは協調磁界の変化がもっとも少ない場所です．測定のためには，励磁信号発生回路の位相オフセット・レジスタを，対の一方では45°進ませ，他方で 45°遅らせる設定にして，協調磁界と差動磁界を発生させせます．これにより，式（14）の状態になります（運用時と同等）．

次に，ピックアップ・コイルで X， Y， Z 各軸方向の差動磁界の強度，具体的には，各磁界の 3コイルでの検出値の 2乗和の平方根である，$\sqrt{R_{i,1}^2+R_{i,2}^2+R_{i,3}^2}$を測定します．この値が 3軸とも最小になる点を探します[注30]．差動磁界の強度は中央で0になるため，3軸すべてで十分小さな強度を示せば，ピックアップ・コイルがほぼ中央にあります．そこで，今度は協調磁界の強度を測定します．協調磁界と差動磁界は一括して出力しているため，この測定値で両者の振幅が確定します．

以前のシステムでは，目分量で中央を探したり，生の測定値を見ながら調整していて使い勝手が悪かったので，今回のシステムでは校正方法も考えつつ，システム設計を行いました．



モーション・キャプチャの応用例

以上のシステムの応用例として，“バーチャルはえたたき”と“バーチャル物体搬送”を試作しました．いずれも，モーション・キャプチャの関数 `GetMagcap()`を呼び出して，ピックアップ・コイルがついたグリップの位置と姿勢を取得し，それに基づいて画面をDirect3D（DirectX）によって描画しています．

● バーチャルはえたたき

この例では，グリップから仮想的に伸びた“はえたたき”を振り回し，“はえ”[注31]をたたきます．ヒットすると，“はえ”ははじき飛ばされます．空間内の要素は 2点，“はえ”と“はえたたき”です．

注 28：UnivIF の動作クロックは，PCI バスから受け付るクロックを PLL で 66.6MHz にしたものであるため，微妙な位相ずれはあるものの，ほぼ同期している．

注 29：対になるコイルで位相レジスタは同期している．

注 30：画面にレベル・メータを表示しておき，それを見ながら人手で行う．慣れればすぐに見つかる．

注 31：本来はデザイン上は“蚊”だったのだが，最終的にユーザ・インターフェースがはえたたきになってしまったので，便宜上“はえ”といっている．

“はえ”は，その速度のx，y，z成分を適当な周期の三角関数で与えています．三角関数は積分しても三角関数であるため，位置も基本的には三角関数であらわされ，3次元のリサージュ図形を描くようにしています注32．あえて速度で運動を規定する理由は，位置調整のしやすさにあります．ヒットしてはじき飛ばされた“はえ”は，いったん遠くに行ってしまいますが，それを元の場所に戻さなければなりません．そこで，座標$\boldsymbol{p}$を速度vから以下の式により計算します．

$$\boldsymbol{p}[k]=\boldsymbol{p}[k-1]\times 0.99+v\Delta t \cdots\cdots(30)$$

この式で“×0.99”の部分が重要で，原点から離れていると，徐々に大きさが小さくなってくる＝戻ってきます．後半は速度を数値積分して位置にしています．これと同じ式を位置だけで処理しようとするとやっかいなので，速度指定にしました．なお，“はえ”の姿勢は，曲がっている軌道が瞬間的に乗っている平面を基準に求めています(“はえ”基準で水平旋回して見える)．

“はえたたき”の表示は，そのままモーション・キャプチャの情報を用いています．両者の衝突判定は，“はえ”の位置を“はえたたき”の座標軸上に変換し，

- “はえ”が“はえたたき”の面を横切ったか
- 横切った点は叩く面にあるか

を判定します．

静止画ではわかりにくいのですが，図21にその動作写真を示します．図(a)では，握られたグリップの姿勢と画面の中の“はえたたき”の姿勢が一致しています．図(b)では，仮想的な“はえたたき”を振り回したときの画像です注33．

これが何の役に立つか，というと何の役にもたたない気がするのですが，応答性の検証という意味ではそれなりに楽しめました．ただ，3次元で動き回る物を2次元のディスプレイで表示しているため，奥行き感覚をつかみにくく，なれるまではなかなかたたけませんでした注34．

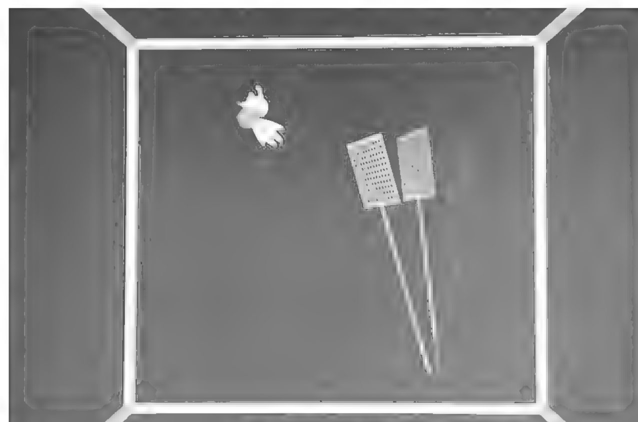
● バーチャル物体搬送

モーション・キャプチャのグリップは，握ったことを検出できるセンサになっています．握る部分に，2枚のアルミ箔でサンドイッチ状態にしたIC保管用の導電スポンジを巻いて，外に保護用にもう一枚のスポンジ(図10で見えているのはこの部分)を巻いています注35．導電スポンジは普段は高抵抗ですが，つぶすと一気に抵抗値が下がるという特性があり，このグリップでは両アルミ箔間の抵抗値が握りぐあいによって変化します．非常に単純で安価な圧力センサです．

これを適当なUSB接続のゲーム・コントローラのボタンに並列に接続して，DirectInputを用いて，ボタンのON/OFFとして値を読み取りました注36．



(a) モーション・キャプチャ内で仮想のはえたたきを振る



(b) 画面のようす

図21 バーチャルはえたたき

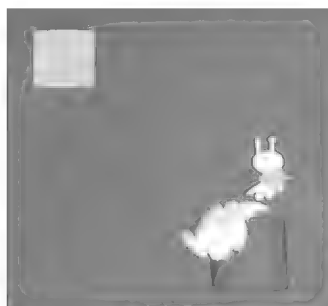
モーション・キャプチャで読み取った位置，姿勢から，仮想空間内でハンドを動かします．ハンドはグリップが握られると閉じます．閉じたときに近くに物体があった場合，閉じている間はその物体をハンドといっしょに移動します．グリップを離したら放置します．物体をハンドといっしょに移動するには，

- 前回と今回のモーション・キャプチャのデータから，ハンドの移動方向，姿勢の変化を算出する
- 連動している物体の位置，姿勢をそのハンドの差分で変更する

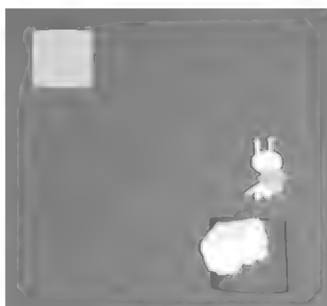
としています注37．ハンドと物体を結合してしまうわけではなく，ハンドの移動と一致した移動を物体にさせるわけです．

実際の動作のようすを図22に示します．図(a)は画面の中のようすで，図(b)は実際の風景です．物体に手を伸ばし，グ

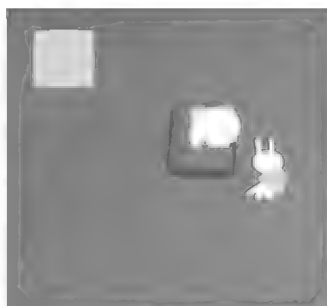
注32：どこか遠くに行ってしまわない関数なら何でもよいのだが，三角関数は手ごろ．
注33：残像があるのは，使用したビデオ・カメラの特性のようである．
注34：原理からすると，空間中央でひたすら振っていればいいのだが :-)．
注35：内側から同心円状に，芯，アルミ箔1，導電スポンジ，アルミ箔2，保護スポンジ，となっている．
注36：最初は増幅回路とかコンパレータなどが必要かと思ったが，予想外にも直結で問題なく動作した．抵抗値の変化がコントローラにちょうど良かったようだ．大きなマットで作れば，運動になるコントローラが作れるかもしれない．
注37：数学的には比較的簡単な座標変換だが，その解説をするとまとまらなくなるので，詳細には触れないでおく．



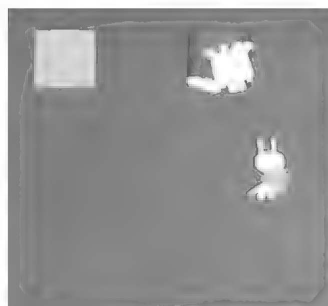
手を伸ばす



にぎる

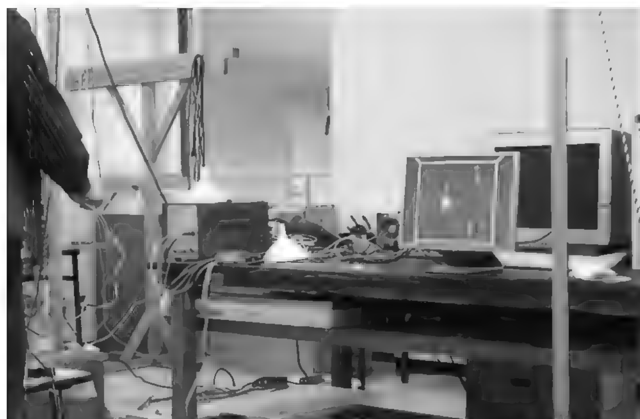


運ぶ



手を離す

（a）画面の中の3次元空間で物体を運ぶ



（b）モーション・キャプチャ内で仮想の物体を運ぶ

図22　バーチャル物体搬送

リップを握るとハンドが閉じます．そのまま手を動かすとハンドも物体といっしょに動きます．目的の場所でグリップを離すとハンドは開き，物体はそこで止まります．

この例はいろいろと使い道がありそうです．そのうち，“ロボットに手取り足取り作業を教える”実験をしてみたいと考えています[注38]．

おわりに

開発したモーション・キャプチャ装置について，その原理と，実装のための信号処理法，具体的な実装について述べました．手段がないため[注39]詳細な評価はしていませんが，これまでの研究結果や基礎的なデータから類推される性能は，1900mmの励磁コイルの中央8割程度の空間全体で，角度分解能0.01度，誤差0.5度，位置分解能0.2mm，精度5mm程度が期待されています．

また，この装置を使った例として，「バーチャルはえたたき」と「バーチャル物体搬送」を紹介しました．人間は，絶対的な精度は案外甘いのですが，相対的な動きや遅れには敏感です．これらの例では本装置の強みである，分解能と応答速度が生かされ，その点では違和感を感じませんでした．はえたたきは楽しい（？）だけですが，物体搬送は今後の応用先がありそうです．

加えて，今回のシステムは磁界を使ったモーション・キャプチャでしたが，間を音にすれば精密な距離測定ができるかもしれないなど，処理部分はほかにも応用ができるのではないかと思っています．

最後になりましたが，本装置の中核部分は東北大学大学院在籍時から開発を行っています．教授の江村超氏，技官の鈴木正俊氏，研究を手伝ってくれた卒業生諸氏，さらに東北学院大学移籍後にお手伝いいただいた機械工場の方々にこの場を借りてお礼申し上げます．また，（財）中谷電子計測技術振興財団の助成を受けて開発が進んだこと，本文執筆中の問い合わせがきっかけで今後の研究用にと特別製のコイルを橋本電気株式会社よりご提供いただいたことを記し，謝意を表します．

注38：ちなみに，SFの世界では3次元の入力装置がよく出てくるが，はっきりいって常用に耐えない．これらの応用例も腕を上げたまま5分も遊んでいると，腕が重くなっていやになってくる．無重力空間で試してみたいものだ．

注39：分解能が高くなり，サイズも大きくなったため，手持ちの機材で評価できなくなった．

参考文献

（1）江村超，熊谷正朗，野村亮太；回転磁界と差動磁界を用いたモーション・キャプチャの開発，計測自動制御学会論文集，Vol.38 No.2，pp.129-136，2002年

（2）Masaaki Kumagai，Takashi Emura；Development of a Motion Capture System That Uses Alternating Field，Proc. of SICE2002，pp.WA16-3，2002年

（3）熊谷正朗；PC用多目的入出力ボードの開発とその応用，日本機械学会ROBOMEC03予稿集，1A1-3F-E7，2003年
http://www.mech.tohoku-gakuin.ac.jp/rde/contents/digital/univif/indexframe.html

（4）三上直樹；原理から学ぶディジタル信号処理技術，インターフェース 2004年9月号，pp.49-64，CQ出版（株）

（5）宮田浩，わかる！コイルと磁気と回路の世界，トランジスタ技術 2004年8月号 別冊付録，CQ出版（株）

（6）熊谷正朗；タイミング・チャート清書ツール Tchart，トランジスタ技術 2002年11月号，pp.204-206，CQ出版（株）
http://www.mech.tohoku-gakuin.ac.jp/rde/contents/library/tchart/indexframe.html

くまがい・まさあき　東北学院大学工学部

Appendix

第5章を読み進むにあたっての補足事項

熊谷 正朗

● 三角関数の基本公式

本手法では三角関数の性質をいろいろ使います．ここで関連するものを簡単におさらいしておきます．

$$\sin(A+\frac{\pi}{2})=\cos A,\ \ \cos(A+\frac{\pi}{2})=-\sin A$$

$$a\ \cos A+b\ \sin A=\sqrt{a^2+b^2}\sin\left(A+\tan^{-1}(\frac{a}{b})\right)$$

$$\sin(A+B)\ =\sin A\cos B+\cos A\sin B$$

$$\cos(A+B)\ =\cos A\cos B-\sin A\sin B$$

$$\sin A\sin B\ =-\frac{1}{2}\left[\cos(A+B)-\cos(A-B)\right]$$

$$\sin A\cos B\ =\ \frac{1}{2}\left[\sin(A+B)+\sin(A-B)\right]$$

$$\cos A\sin B\ =\ \frac{1}{2}\left[\sin(A+B)-\sin(A-B)\right]$$

$$\cos A\cos B\ =\ \frac{1}{2}\left[\cos(A+B)+\cos(A-B)\right]$$

$$\frac{d}{dx}\sin(ax)\ =a\cos(ax),\ \ \frac{d}{dx}\cos(ax)=-a\sin(ax)$$

● ベクトル

座標や方向を扱うためにベクトルを使用します．本章では太字で表記しました．

● ベクトルの利用

空間での位置，もしくは空間での方向を表すためにx，y，zの3要素からなるベクトルを使用します．ベクトル$\boldsymbol{v}$に対して，その要素はv_x，v_y，v_zと表記します．単に方向を示す場合はベクトルの大きさは自由ですが，本文では"方向ベクトル"といった場合は暗黙に長さ1を仮定します．

長さを1にするには，

$$\ell\ |\boldsymbol{v}|=\sqrt{v_x^2+v_y^2+v_z^2}$$

$$v'_x\ =\frac{v_x}{\ell},\ \ v'_y=\frac{v_y}{\ell},\ \ v'_z=\frac{v_z}{\ell}$$

によって計算します．

● ベクトルの内積

二つのベクトル$\boldsymbol{a}$，$\boldsymbol{b}$に対して，

$$\boldsymbol{a}\cdot\boldsymbol{b}=a_x b_x+a_y b_y+a_z b_z$$

で計算される値をベクトルの内積といいます．ベクトルがなす角度をθとすると，

$$\boldsymbol{a}\cdot\boldsymbol{b}=|\boldsymbol{a}|\ |\boldsymbol{b}|\ \cos\theta$$

とも表せます．

ベクトルの内積を用いると，ベクトルのなす角が得られます．とくに，内積が0の状態はベクトルが直交することを示すためによく使われます．

● ベクトルの外積

二つのベクトル$\boldsymbol{a}$，$\boldsymbol{b}$に対して，

$$c_x=a_y b_z-a_z b_y$$

$$c_y=a_z b_x-a_x b_z$$

$$c_z=a_x b_y-a_y b_x$$

で定義されるベクトル$\boldsymbol{c}$をベクトル$\boldsymbol{a}$，$\boldsymbol{b}$の外積，$\boldsymbol{a}\times\boldsymbol{b}$といいます．外積の特徴は，それをなすベクトルと直交する，すなわち，$\boldsymbol{a}\cdot\boldsymbol{c}=0$，$\boldsymbol{b}\cdot\boldsymbol{c}=0$が成り立つことです．ベクトルの長さは，$|\boldsymbol{a}|\,|\boldsymbol{b}|\sin\theta$となります．2本のベクトルに直交する線や，平面の法線を求める際によく使います．

● ベクトル場

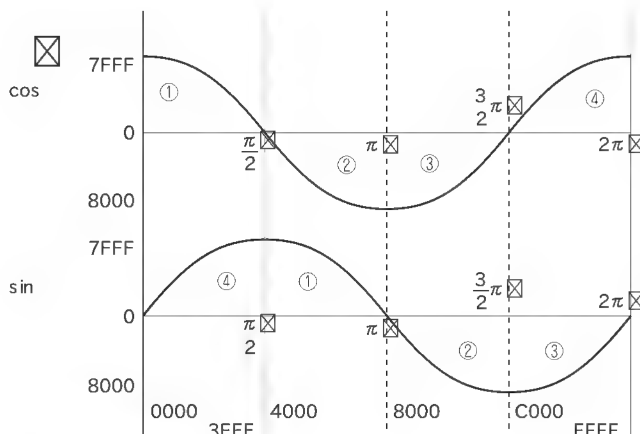
空間内のすべての点で，その点での方向を表すベクトルなどを定義するものがベクトル場です．たとえば，気象情報の日本全土の風向，風速のようなものです．アメダスでは測定地点の情報しかありませんが，実際にはいたるところで瞬間ごとに風の強さと方向が分布しています．本文中の磁束密度場はベクトル場で，空間の各点における磁束密度（磁場の強さ）の方向と強さをベクトルとして表しています．

● 三角関数参照回路の構成

本装置で使用した三角関数の参照回路は，数表を書き込んだ32Kバイト（×8，15ns）の高速SRAMとロジック回路で構成しました．入力は0～2πの範囲を16ビットで表し，出力は16ビットです．

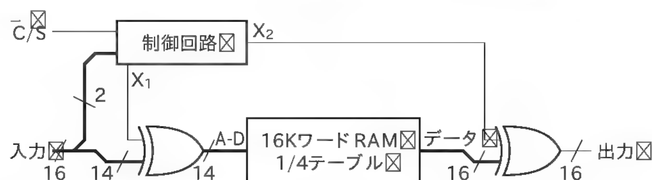
データは上位，下位を順次読み込むことにしますが，16Kワード（2^{14}）しかありません注A．そこで，三角関数の特性を利用することにしました．

三角関数は図Aに示すように，余弦，正弦ともπ/2ごとに，0～π/2の範囲の余弦関数を上下左右に折り返して，すなわち正負を反転させて作ることができます．数表のSRAMでいえば，上下の反転



図A 三角関数の対称性

注A：もちろん，64Kワード分（1Mビット）のSRAMを用意すればこのような手法は不要．ただし，アドレスとデータ・バスをあわせて24ビットを超えることが時に制約になる．また，256Kビット品は一時期CPUの外付けキャッシュとして量産されていたので手ごろ．



図B　三角関数生成回路のブロック図

表B　三角関数発生制御回路の真理値表

入力		Cos				Sin			
I_{15}	I_{14}		C/S	X_1	X_2		C/S	X_1	X_2
0	0	①	0	0	0	④	1	1	0
0	1	②	0	1	1	①	1	0	0
1	0	③	0	0	1	②	1	1	1
1	1	④	0	1	0	③	1	0	1

※X_1＝1：アドレス反転　X_2＝1：データ反転

は出力データの反転，左右の反転はアドレスの反転にあたり，状況に応じて図Bに示すようにXORを用いて反転させました[注B]．制御部の真理値表を表Bに示します．

ここで，2進数の正負反転について検討します．本来2進数で数値を表すときは，表Cに示す，2の補数表記を用います．この表記法では，数値の正負を反転させるには，"全ビットNOTして，1を加算"する必要があります．つまり，正負反転には加算器が必要です．そこで，SRAMに書き込む数表を作る際に，"＋0.5"補正を行いました．具体的には，表Dに示すように，2進数と10進数を対応づける際に，10進数では2の補数表記に"＋0.5"したものを表すと規定します．この定義においては，全ビットをNOTするだけで，負号反転を行うことができ，しいては回路の簡素化，高速化が可能となります．

この表記は関数の横軸にも採用しています．この処理を行わない場合には関数を得るために0～16384の16385個の数値が必要になります．"ジャスト・ゼロ"を捨てることが，常日頃もっていた"なぜ正の数より負の数は一個多いんだ"という不満の解消（？）にもなりました．

● ディジタル・フィルタの特性

ディジタル・フィルタについては本誌2004年9月号の特集で詳細に解説されているので，ここではそれに補足するにとどめます．ディジタル・フィルタはFIR型とIIR型の二つがあり，先の特集ではFIR型について解説されていました．

FIR型は，その出力は直前のいくつか（膨大なこともあるが）の入力のみによって決まるのに対して，IIR型では直前の入力に加え，直前の出力に依存します．別の見方では，FIR型ではその伝達関数がzの多項式によって表されますが，IIR型は分数式で表されます．

FIR型は先の特集で解説されていたように，急峻な特性のフィルタを数学的に設計しやすく，また遅延が一定しているため，音声処理などに適しています．その反面，サンプリング周波数とカットオフ周波数が大きく離れる場合に膨大な遅延素子や演算時間が必要になったり，位相遅れが大きくなりやすく，フィードバック・ループに入れにくいという難点もあります．一方，IIR型はほぼ逆の性質を持ちます．設計はある程度トライ＆エラーで，急峻な特性も得にくいのですが，少ない演算で低いカットオフを実現しやすく，その分位相も遅れにくいという利点があります[注C]．筆者は制御系の出身でもあり，IIR型のほうが好みです．

IIR型を数式で見ると，次のようになります．

$$y[k]=-d_1y[k-1]-d_2y[k-2]+\cdots+d_ny[k-n]$$
$$+c_0u[k]+c_1u[k-1]+\cdots+c_nu[k-n]$$

$y[k]$を求めるために，$y[k-i]$が必要になります．これをz変換して，伝達関数を求めると，

$$H(z)=\frac{c_0+c_1z^{-1}+\cdots+c_nz^{-n}}{1+d_1z^{-1}+\cdots+d_nz^{-n}}$$

となります．たとえば，本文で使用した，

$$y[k]=(1-r)y[k-1]+ru[k]$$

の場合，伝達関数は，

$$Y(z)-(1-r)Y(z)z^{-1}=rU(z)$$
$$H(z)=\frac{Y(z)}{U(z)}=\frac{r}{1-(1-r)z^{-1}}$$

となります．

このIIR型フィルタにはもう一つ，アナログ・フィルタの設計をある程度流用できるという特徴があります．具体的には，アナログ・フィルタの伝達関数$H(s)$に対して，双一次変換，

$$s=\frac{2}{T}\frac{z-1}{z+1}$$

を代入し，$H(z)$を求めます（Tはサンプリング周期）．得た伝達関数から，演算式を復元して実装すると，よく似た特性のフィルタが再現できます[注D]．

IIR型フィルタの伝達特性は，FIR型と同様，$z=e^{sT}$，$s=j\omega=2\pi jf$

表C　2進数による一般的数値表記

A_{16}	A_{10}	A_{16}	A_{10}	$(A+1)_{10}$
0x7f	127	0x80	－128	－127
0x02	2	0xfd	－3	－2
0x01	1	0xfe	－2	－1
0x00	0	0xff	－1	0
0xff	－1	0x00	0	1
0xfe	－2	0x01	1	2
0x80	－128	0x7f	127	(*)128

(*)8ビットの場合溢れて－128に戻る

表D　2進数による"＋0.5"表記

A_{16}	A_{10}	A_{16}	A_{10}
0x7f	127.5	0x80	－127.5
0x02	2.5	0xfd	－2.5
0x01	1.5	0xfe	－1.5
0x00	0.5	0xff	－0.5
0xff	－0.5	0x00	0.5
0xfe	－1.5	0x01	1.5
0x80	－127.5	0x7f	127.5

注B：XORはその定義より，ビットを反転させるかさせないか，という使い方ができる．
注C：また別の見方をすると，FIR型の周波数特性は周波数軸がリニアで，IIR型は対数で表記されることが多いようである．
注D：ただし，サンプリング周波数とカットオフ周波数が離れ，かつもとの伝達関数の分母の次数が高いと計算精度が不足しがちになる．また，サンプリング周波数/2に近づくほど特性がずれる．

リスト A　伝達特性計算

```
#include "stdafx.h"                 // VC++でのみ必要
#define _USE_MATH_DEFINES           // VC++
#include <complex>
#include <stdio.h>
#include <math.h>

using namespace std;                // VC++
typedef complex<double> c_complex;

// 伝達特性算出のためのパラメータ
const double SamplingF=66.6e6/256;//サンプリング
const double T=1/SamplingF;       //周波数/周期
const double MinF=10;             //下限周波数
const double MaxF=10000;          //上限周波数
const int NumStep=1000;           //周波数分割数

int main(void)
{
  double f,w;                     // 周波数,ω=2πf
  c_complex s;                    // s=jω
  c_complex z,zi;                 // z=exp(sT) zi=1/z;
  c_complex tf;                   // 伝達関数
  double gain,phase;              // ゲイン，位相

  for(int i=0;i<=NumStep;i++)
    {
      f=MinF*pow(10, log10(MaxF/MinF)*i/NumStep);
      // f=MinF+(MaxF-MinF)*i/NumStep;  // リニア

      w=2*M_PI*f;
      s=c_complex(0,w);                     // s=jw
      z=exp(T*s);  zi=c_complex(1)/z;

      // 伝達関数
      //tf=1.0/(1.0+0.01*s);                // 's'のままでも可
      double r=1.0/1024;   tf=r/(1.0-(1.0-r)*zi);
      //for(int k=0,tf=0;k<1024;k++) tf+=pow(zi,k);
      //tf/=1024.0;

      gain =abs(tf);  phase=arg(tf);
      printf("%g, %g, %g,    %g, %g¥n",
             f,log10(gain)*20,phase*180/M_PI,
             gain,phase);
    }
  return 0;
}
```

リスト B　磁束密度の計算（円電流）

```
// 円電流分割単位角度
const double DIR_DELTA=2*M_PI/40;  // [rad]
// 円電流の磁場
void CalculateCircleCurrent(
                 double x,y,z,      // 観測点
                 double cx,cy,cz,   // 中心点
                 double nx,ny,nz,   // 法線
                 double Rd,  I,     // 半径,電流
                 double &bx,&by,&bz)// 磁束密度
{
  // 円を構成する２軸ベクトル
  double r1x,r1y,r1z,r2x,r2y,r2z;
  // 円のためのベクトルを生成(円の平面，直交)
  // ベクトルr1をnと垂直になるように生成
  if(hypot(nx,ny)<1e-10) { r1x=0; r1y=nz; r1z=-ny; }
  else                   { r1x=ny; r1y=-nx; r1z=0; }
  // 以上で n・ r1 = 0 → 垂直
  r2x= ny*r1z - nz*r1y;                // r2=n×r1(外積)
  r2y= nz*r1x - nx*r1z;
  r2z= nx*r1y - ny*r1x;
  // ここで n・r2=r1・r2=0
  // r1, r2 が長さRdになるように調整
  double r;
  r=Rd/sqrt(r1x*r1x+r1y*r1y+r1z*r1z);
  r1x *= r;  r1y *= r;  r1z *= r;
  r=Rd/sqrt(r2x*r2x+r2y*r2y+r2z*r2z);
  r2x *= r;  r2y *= r;  r2z *= r;

  int j,li;
  // 分割数決定  まず線素長で評価.
  //                最低でも円弧分割数以上.
  li=(int)(2*M_PI*Rd/LEN_DELTA)+1;  // 円周長さ
  if(li<2*M_PI/DIR_DELTA)
    li=(int)(2*M_PI/DIR_DELTA);     // 円弧角度

  r=(2*M_PI*Rd/li)  /Rd;
  // rは線素補正値 前半: 円弧長  後半: r1,r2長
  for(j=0;j<li;j++)
  {
    double c=cos(2*M_PI*j/li);
    double s=sin(2*M_PI*j/li);
    Calculate_DS(
       x-(cx+r1x*c+r2x*s), y-(cy+r1y*c+r2y*s),
       z-(cz+r1z*c+r2z*s),            // 円上の位置
       r*(-r1x*s+r2x*c), r*(-r1y*s+r2y*c),
       r*(-r1z*s+r2z*c),              // 線素ベクトル
       I,bx,by,bz);
  }
}
```

を代入することで得られ，$H(f)$の絶対値がゲイン，偏角が位相となります．

最後に，リストAに本文のフィルタ特性を求めるために使っている[注E]プログラムを示します．Microsoft Visual C++.NETおよびRedHat Linux 9で動作します[注F]．伝達関数"tf="を書き換えれば，ほかの用途にも使えると思います．

● 磁束密度の計算

円電流により発生する磁束密度の計算例についても紹介します（リストB）．円電流の位置を規定するために，円の中心と，コイル面の法線ベクトルを採用してみました．

狭い間隔で複数並べるとソレノイドの磁束密度などもわかります．また，半径を十分に小さく（＆電流大きく）すると，ほかの磁気式モーション・キャプチャで採用されていたりする磁気双極子を模擬することもできます．

実は当初，本方式は立方体の頂点８箇所に小型の円筒励磁コイルを置く形が検討されていたのですが，計算の結果，磁束密度場の変化がより直線的な現在の方法に変更しました．

くまがい・まさあき　東北学院大学工学部

注E：実は，本文の図にはもう少し周波数選定に凝ったプログラムを使っているので，おそらく同じ結果は得られない．
注F：結果を適当なCSVファイルにリダイレクトして，グラフ化してほしい．

P 1 2 3 4 5 Ap 6

Chapter 6

センサの選定と計算が重要になる

フィードバック制御のために必要となる計測

本章では，フィードバック制御のデータ計測について考える．制御ではフィードバックが基本だが，すべての状態量が制御に必要というわけではない．制御対象の特性を見きわめ，それに応じたデータをセンサで計測し，システム設計に活用することになる．また，式の展開によってセンサでは測ることができない数値も割り出すことができる．本章ではその一例を示す．　（編集部）

川谷 亮治

手の上に棒や傘などを立たせる遊びを一度は経験したことがあると思います．床の上に立てた棒を少し傾けた状態で離すと当然ですが倒れます．しかし，棒を手の上において，手をうまく動かせば棒は倒れなくなります（図1）．

ここでうまく動かせばということが大切です．適当に動かしただけではだめ，ということは容易にわかります．そして，そのため（うまく動かすため）には，棒の現在の状態を知ることが重要になります．

このように，制御したい対象（制御対象）の現在の状態に基づいて対象に操作を加える制御方式をフィードバック制御といいます．フィードバック制御というとなにやら難しそうだ，と思ってしまう方もいるかもしれませんが，車を運転する，自転車に乗る，物をつかむ，など皆さんの日常の行動を考えたとき，必ずといってよいほどフィードバック制御を知らぬ間に活用しています．実は非常に身近な存在なのです．いろいろな例を考えてみるとよいかもしれません．

ところで，手の上で棒を立たせる場合，皆さんは棒の何を見ていますか？　棒の傾きだけですか？　それとも…？　人間が行ういわゆる手動制御と違って，フィードバック制御をコンピュータを利用して自動的に行わせようとする，いわゆる自動制御を実現する場合，センサを利用して対象の現在の状態を計測する必要があります．何を計測するかによって，使用するセンサが異なります．

したがって，目的を達成する（たとえば，棒を立たせる）制御系を構成するためには，何を計測する必要があるのかを知ることは非常に重要になります．本章では，線形制御理論の立場で，このことについて考えてみたいと思います．なお，本文中の解析や数値シミュレーションはMATLABを使用して行っています．



図1　だれもがやったことのある棒立て

状態フィードバック制御

● 振子系とは

話を簡単にするために，図2に示すような，点Oを中心として自由に回転できる棒（以下では，振子と呼ぶ）を考えます．また，点Oまわりに操作量としてトルクτを直接与えることができるとします．本系を振子系と呼ぶことにします．

これに対して，鉛直上方を振子の角度θの原点にとり，時計回りを正方向とすると，振子に作用するトルクのバランスから

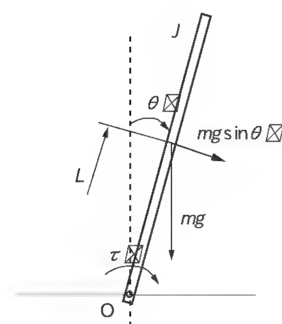


図2　振子系

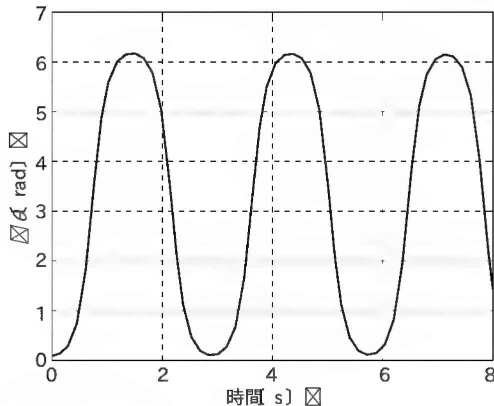


図3　初期値応答

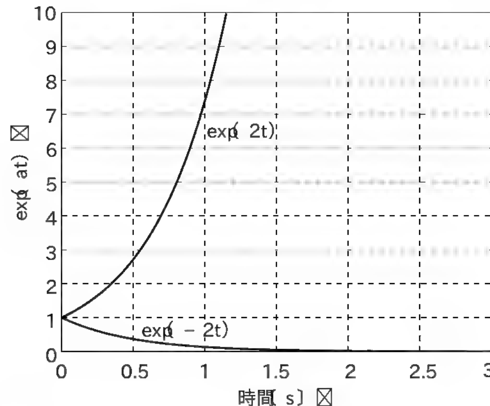


図4　e^{2t}とe^{-2t}

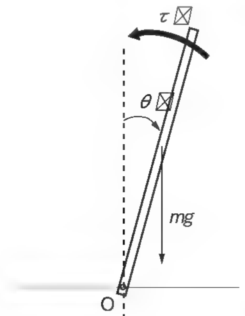


図5　振子を立たせるためには

次に示す微分方程式が簡単に得られます．

$$J\ddot{\theta} = mgL\sin\theta + \tau \quad \cdots\cdots (1)$$

ここで，Jは点Oまわりの振子の慣性モーメント，mは振子の質量，Lは点Oから振子の重心までの距離，gは重力加速度です．なお，振子が回転運動することよって生じる粘性抵抗はないものと仮定します．たとえば，$mgL/J = 36$として，トルクを与えない($\tau = 0$)で初期条件$\theta(0) = 0.1$[rad]，$\dot{\theta}(0) = 0$に対して数値シミュレーション(初期値応答)を行った結果を図3に示します．横軸が時間([s])，縦軸が振子の角度([rad])を表しています．図より，振子が鉛直下方($\theta = \pi$[rad])を中心に単振動していることがわかると思います(ここでは，空気等の粘性抵抗を考慮していないので，振幅が一定の振動を続けることに注目してほしい)．

ところで，式(1)は非線形特性(右辺第1項)をもちますが，振子の角度θが微小である(おおよそ$|\theta| < 30°$の範囲)と仮定すると，$\sin\theta \approx \theta$と近似できます．以降では，このように近似した線形微分方程式，

$$J\ddot{\theta} = mgL\theta + \tau \quad \cdots\cdots (2)$$

を議論の対象とします．

上式に対して，$\tau = 0$としたときの解θは，

(a) $\theta = e^{\lambda t}$と仮定し，線形微分方程式(2)に代入

(b) そのとき得られる方程式$\lambda^2 - mgL/J = 0$の解を求める($\lambda_{1,2} = \pm\sqrt{mgL/J}$)

ことで，

$$\theta = C_1 e^{\alpha t} + C_2 e^{-\alpha t} \quad \cdots\cdots (3)$$

で与えられます．ここで$\alpha = \sqrt{mgL/J}$ (>0)であり，C_1，C_2は初期条件$\theta(0)$，$\dot{\theta}(0)$から決定される定数です．この結果から，$\tau = 0$のとき，(右辺第1項の$e^{\alpha t}$により)θはどんどん大きくなる(図4)，つまり不安定であることがわかります[なお，式(2)はθが微小な範囲しか適用できないことに注意]．

いま，振子系に対して，振子を鉛直上方に立たせる($\lim_{t\to\infty}\theta = 0$となるような)制御を行うことを主たる制御目的とします．そのためのトルクτの与え方について考えてみましょう．

● 制御方策(比例制御)を考える

振子を立たせるためには，振子の現在の角度θに対して，その逆方向にトルクを与えればよい，というアイデアが安直に思いつくかもしれません(図5)．

そのアイデアをもっとも簡単に実現する例として，$\tau = -K_p\theta$とします(振子の角度に比例してトルクを与えるということで，これを比例制御という)．ここでK_pは角度θからトルクを決定するための比例定数です．なお，少なくとも，重力によって倒れようとする振子を倒れないようにするために十分なトルクを与える必要があるので，$K_p > mgL$とします．これを式(2)に代入して少し変形すると次式を得ます．

$$J\ddot{\theta} + (K_p - mgL)\theta = 0 \quad \cdots\cdots (4)$$

前述と同様の手順で微分方程式(4)を解き，式変形を行うと，

$$\begin{aligned}\theta &= C_1 e^{j\beta t} + C_2 e^{-j\beta t} \\ &= C_1(\cos\beta t + j\sin\beta t) + C_2(\cos\beta t - j\sin\beta t) \\ &= (C_1 + C_2)\cos\beta t + (j(C_1 - C_2))\sin\beta t \\ &= A\sin(\beta t + \phi) \quad \cdots\cdots (5)\end{aligned}$$

を得ます．ここで，$\beta = \sqrt{(K_p - mgL)/J}$ (>0)．なお，式変形の際にEulerの公式($e^{jx} = \cos x + j\sin x$)を利用しています．また，$A$，$\phi$は初期条件によって定められる定数です．上式は，振子が$\theta = 0$を中心として単振動を行うことを意味しています．つまり，θを利用したフィードバック制御$\tau = -K_p\theta$により確かに振子が倒れなくはなりますが，$\lim_{t\to\infty}\theta = 0$とすることは残念ながらできません．

● 適度な粘性を与える方法

図3のシミュレーションでは振子が単振動を行っていたことを思い出してください．ところが，同様のことを実験で確かめると，振子は必ず最終的に静止します．それは，実際には空気

などの粘性抵抗が存在するからです．ここが重要な点です．粘性抵抗が作用すれば振動は減衰するのです．

ところが，今，考えている振子系においては，粘性抵抗は考慮していません．この粘性抵抗により生じる力（トルク）は，速度に比例した力（トルク）なので，それをフィードバック制御を利用して等価的に与えることにします．つまり，

$$u = -K_p\theta - K_v\dot{\theta} \quad \cdots\cdots (6)$$

です．右辺第2項目が粘性抵抗を与えるために比例制御に付加した項です．これを式（2）に代入すると，

$$J\ddot{\theta} + K_v\dot{\theta} + (K_p - mgL)\theta = 0 \quad \cdots\cdots (7)$$

を得ます．この微分方程式の解もこれまでと同様の手順で解けますが，かなり複雑な結果となるので，ここでは具体的に解を示すことは省略し，初期値応答（$K_v/J = 5$，$(K_p - mgL)/J = 36$，$\theta(0) = 0.1$[rad]，$\dot{\theta}(0) = 0$[rad/s]）のみを示します（図6）．

ここで重要な点は，$K_v>0$，$K_p>mgL$ の条件を満たすように K_p，K_v が選定されていれば $\lim_{t\to\infty}\theta = 0$，つまり振子が鉛直上方に立つことです．このような性質を漸近安定といいます．また，式（6）の制御則を状態フィードバック制御と呼ぶことにします．つまり，振子系に対しては，状態フィードバック制御により漸近安定にできます．

状態フィードバック制御の実現

状態フィードバック制御の式（6）において，K_p，K_v の与え方によって，振子が鉛直上方に立つ，つまり $\theta = 0$ となるまでに生じる応答は異なります．したがって，設計者はこれらを適切に定める必要があります．

一方，それを実現するコンピュータは，式（6）の右辺の計算を行い，得られた操作量 u を制御対象に与えるという作業手順をまとめたプログラムに基づいて行動します（図7）．

そのためには，振子の現在の（鉛直上方に対する）角度 θ と角速度 $\dot{\theta}$ が必要になります．振子の角度 θ に対しては，ポテンショ・メータやロータリ・エンコーダなどの角度センサを利用して比較的容易に計測できます．前者はA-D変換器を，後者はカウンタ機能を利用することでコンピュータが入手できます．もし，角速度 $\dot{\theta}$ を計測するセンサを利用できるならば，状態フィードバック制御の実現は簡単なのですが，（角）速度用のセンサは意外に少ないのです．

これに対して，振子系では角度 θ が計測できるのだから，それをアナログ的あるいはディジタル的に微分して角速度 $\dot{\theta}$ を求めればよい，と考えるかもしれません．それで悪いというわけではありませんが，前者に対しては観測ノイズの処理，後者に対してはコンピュータの有限語長による桁落ちの問題に配慮しなければなりません．

そこで，本章では，状態フィードバック制御を実現するための一つの方法として制御対象の状態を推定する状態観測器を紹介します．状態観測器が利用できるためには制御対象が後述する条件を満たさなければなりませんが，それを満たせば，制御対象のもつすべての状態を推定できます．センサで直接計測することができないものでも推定できるのです．フィードバック制御に限らず，いろいろな方面への適用可能性が考えられるかもしれません．

状態空間モデルの数式表現

本章で利用する線形制御理論は，制御対象に対して導出した線形微分方程式を解析，ならびに設計の基本とします．振子系の場合は式（2）です．ただし，その線形微分方程式をそのまま利用するのではなく，状態方程式と呼ばれる形（連立1階線形微分方程式）に変形します．式（2）に対する変形の一例を次式に示します．

$$\dot{x} = \begin{bmatrix} 0 & 1 \\ \alpha^2 & 0 \end{bmatrix} x + \begin{bmatrix} 0 \\ b_1 \end{bmatrix} u$$

ここで，$\quad\cdots\cdots (8)$

$$x = \begin{bmatrix} \theta \\ \dot{\theta} \end{bmatrix}$$

であり，$\alpha^2 = mgL/J$，$b_1 = 1/J$，$u = \tau$ です．

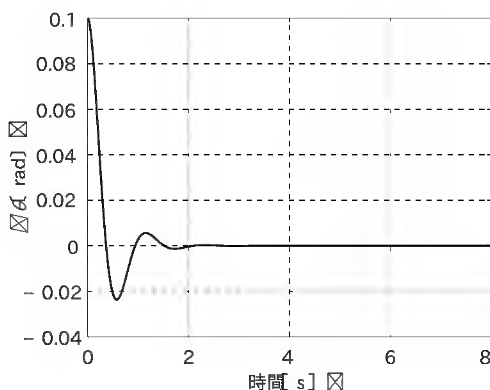


図6
漸近安定な制御系の実現

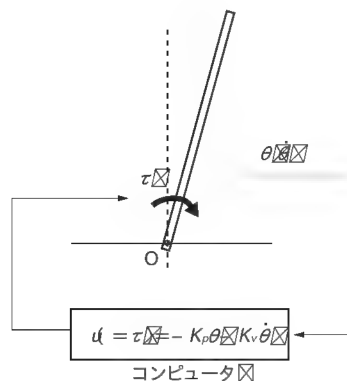


図7
状態フィードバック制御

上式をより一般的に表現したのが，

$$\dot{x} = Ax + Bu \quad \cdots\cdots (9)$$

です．xを状態量と呼びます．A，Bは行列である点に注意してください．状態方程式に関連したいくつかの重要事項をまとめておきます．詳細については本章では省略します．興味ある方は，市販されている制御関連の本を参照してください．

(a) 行列Aのすべての固有値(極)の実部が負であるとき，制御を施していない，つまり$u=0$に対する状態方程式の解xは任意の初期状態$x(0)$に対して，

$$\lim_{t\to\infty} x(t) = 0$$

となる．いわゆる漸近安定である．逆に，一つでも実部が正の固有値をもつと$\lim_{t\to\infty} x(t) = \infty$．

(b) 可制御行列

$$V = [B \ AB \ A^2B \cdots A^{n-1}B]$$

に対して，$rank(V) = n$を満たすとき，状態方程式が可制御であるという．ここで，nは行列Aのサイズを意味する．

(c) 状態方程式が可制御であるとき，状態フィードバック制御

$$u = -Kx \quad \cdots\cdots (10)$$

により，行列$A-BK$の固有値を自由に配置できる．

状態方程式(9)に対して状態フィードバック制御式(10)を施すことで，

$$\dot{x} = (A - BK)x$$

を得ます．(c)は，その固有値が自由に指定可能であることを意味しています．

振子系に対して(a)〜(c)を適用してみましょう．

(a) 行列Aの固有値は$|\lambda I - A| = 0$より，$\lambda_{1,2} = \pm\alpha$．よって，実部が正の固有値をもつので不安定．

(b) 可制御行列は，

$$V = [B \ AB] = \begin{bmatrix} 0 & b_1 \\ b_1 & 0 \end{bmatrix}$$

$rank(V) = 2$であるので可制御．

(c) この場合の状態フィードバック制御は，

$$u = -Kx = -k_1\theta - k_2\dot{\theta}$$

であり，式(7)と一致する($k_1 = K_p$，$k_2 = K_v$)．

状態フィードバック制御式(10)の実現には状態量xの情報が不可欠です．でも，振子系のように，状態量すべてではなく，その一部のみが入手できる立場をとるのが一般的です．振子系の例で観測量yを角度θとすると，

$$y = \theta = [1 \ 0]x \quad \cdots\cdots (11)$$

となります．つまり，観測量を表す観測方程式は状態量に適切な行列Cをかけた，

$$y = Cx \quad \cdots\cdots (12)$$

で与えられます($y = Cx + Du$とするのがより一般的な表現だが，ここでは$D=0$として考える)．

式(9)と式(12)をまとめた，

$$\begin{cases} \dot{x} = Ax + Bu \\ y = Cx \end{cases} \quad \cdots\cdots (13)$$

を状態空間モデルと呼びます．

状態観測器の数式表現

● 対象とする状態空間モデル

線形制御理論に基づく制御器の基本は状態フィードバック制御式(10)です．しかし，センサを通して利用できる情報は観測量のyだけです．そこで，それを利用して直接計測できない状態量を推定することを考えます．その際に，状態量が推定できるかどうかは制御方策(uの決定法)には依存しないので，ここでは$u=0$とした状態空間モデル，

$$\begin{cases} \dot{x} = Ax \\ y = Cx \end{cases} \quad \cdots\cdots (14)$$

を対象とします．制御対象の特性を表す行列Aとセンサに関連した行列Cとの関係が重要になりそうだ，ということは容易に推測できます．

● 状態観測器の考え方

行列Aが既知とすると，コンピュータを利用して，次の微分方程式を数値的に解くことができます(図8)．

$$\dot{\hat{x}} = A\hat{x} \quad \cdots\cdots (15)$$

なお，その際に，初期条件$\hat{x}(0)$を定める必要があります．制御対象のもつ状態量xのすべてを手にできないということは，もちろん初期条件$x(0)$も手にすることはできないので，適当に定めるしかありません．そのため，$x(0) - \hat{x}(0) \neq 0$である点に注意してください．

xと$\hat{x}$の関係を調べるために，$u=0$とした状態方程式($\dot{x} = Ax$)と式(15)を引き算します．

$$\dot{x} - \dot{\hat{x}} = A(x - \hat{x}) \quad \cdots\cdots (16)$$

$\hat{x}$はコンピュータの中で計算されているので，完全に既知で

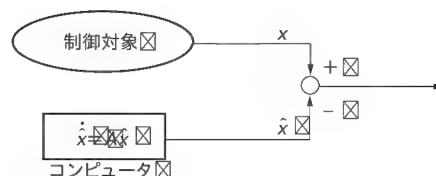


図8 状態観測器の基本

す．もし，

$$\lim_{t\to\infty}\left(x-\hat{x}\right)=0 \text{ つまり } \lim_{t\to\infty}\hat{x}=x \quad \cdots\cdots(17)$$

であるならば，$\hat{x}$をxの代用として利用できそうです．そのためには，式(16)が漸近安定，言い換えると，行列Aのすべての固有値の実部が負であることが必要になります．しかし，行列Aは制御対象自身の特性を表現しているもので，必ずしもそれが漸近安定である保証はありません(少なくとも振子系は不安定)．

ここで，ひとくふうします．観測量yは手にできる情報です．もし，$\hat{x}$がxの推定量であるとすれば$C\hat{x}$は観測量の推定量として考えることができます．ということは$y-C\hat{x}$を利用すれば，観測量の立場から推定がどの程度進んでいるのかを知ることができそうです．そこで，式(15)にそれを補正項として加えます．

$$\dot{\hat{x}}=A\hat{x}+H\left(y-C\hat{x}\right) \quad \cdots\cdots(18)$$

ここで，Hは設計者が定めることが許された行列です．これに対して，式(16)と同様にxとの関係を調べると，

$$\dot{x}-\dot{\hat{x}}=(A-HC)\left(x-\hat{x}\right) \quad \cdots\cdots(19)$$

となります．これより，行列Hを利用して行列$A-HC$を漸近安定にできるならば，式(17)の意味で$\hat{x}$をxの推定量として利用できそうです．

● 可観測性の意味

それでは，行列$A-HC$を漸近安定にできる行列Hが存在するための条件を示しておきます(証明略)．これには可観測性が強く関係しています．

可観測行列

$$W=\begin{bmatrix} C \\ CA \\ CA^2 \\ \vdots \\ CA^{n-1} \end{bmatrix} \quad \cdots\cdots(20)$$

が$rank(W)=n$を満たすとき，可観測であるという．このとき，行列$A-HC$の固有値を任意に配置する行列Hが存在する．ここでnは行列Aのサイズを意味する．

ところで，可観測行列Wですが，それのもつ構造には意味があります．

振子系の場合，観測量yは振子の角度θでした．角度θは時刻tで振子がどの程度傾いているのかを表しており，ポテンショ・メータなどのセンサで直接計測できます．ところで，それを微分すると，角速度$\dot{\theta}$が得られますが，それは振子がこれからどちらの方向にどれだけの大きさで動こうとしているのかを意味してます．つまり，微分という操作により，観測量に含まれているより多くの情報を引き出すことができるのです．ちなみに，観測量yの1階導関数$\dot{y}$，2階導関数$\ddot{y}$は，状態方程式を利用すると，

$$\dot{y}=C\dot{x}=CAx,\quad \ddot{y}=CA^2x$$

ですね．可観測行列中に登場する要素が順に登場することがわかりますか．

つまり，可観測行列は，観測量からどれだけ多くの情報(状態量)を引き出せるのかを表すものなのです．そして，そのランクがnになるということは，すべての状態量を観測量から作り出せることを意味します．なお，可観測行列WにはCA^{n-1}までしか含まれていませんが，それ以上繰り返しても情報を引き出せないことを示すことができるためです．

● 振子系の可観測性を調べる

それでは，振子系に対して可観測性を調べてみましょう．まず，振子の角度θが計測できるとします．この場合，

$$A=\begin{bmatrix} 0 & 1 \\ \alpha^2 & 0 \end{bmatrix},\quad C=\begin{bmatrix} 1 & 0 \end{bmatrix}$$

なので，可観測行列は，

$$W=\begin{bmatrix} C \\ CA \end{bmatrix}=\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

で与えられます．$rank(W)=2$であることは明らかです．つまり，式(18)を利用して状態量を推定できます．もし，角速度センサのみが利用できるとしても，

$$W=\begin{bmatrix} 0 & 1 \\ \alpha^2 & 0 \end{bmatrix}$$

より，可観測性が満たされ，状態量が推定できます．

● 制御器の構造

式(18)の導出の際に，$u=0$としましたが，そうでない場合でも，

$$\dot{\hat{x}}=A\hat{x}+Bu+H\left(y-C\hat{x}\right) \quad \cdots\cdots(21)$$

とすることで，状態方程式($\dot{x}=Ax+Bu$)との関係を調べると，式(19)が得られることを簡単に示せます．そこで，式(21)を状態量すべてを推定するという意味で全状態観測器と呼びます．これを利用して推定した状態量xに基づく状態フィードバック制御，つまり，

$$\begin{cases} \dot{\hat{x}}=A\hat{x}+Bu+H\left(y-C\hat{x}\right) \\ u=-K\hat{x} \end{cases} \quad \cdots\cdots(22)$$

もしくは，

$$\begin{cases} \dot{\hat{x}}=(A-BK-HC)\hat{x}+Hy \\ u=-K\hat{x} \end{cases} \quad \cdots\cdots(23)$$

が線形制御理論により得られる制御器です．なお，コンピュータに実装する場合，後者が利用されます．

倒立振子系に対する設計例

● 倒立振子系を構成する要素

これまで例として考えた振子系は，基本的事項を理解するために，その回転軸まわりに直接与えるトルクτを操作量としまし

た．しかし，手の上で棒を立たせる遊びを考えた場合，振子をのせた手(に相当するもの)を動かすことで振子を立たせるという状況設定が必要です．その点を考慮して，ここでは，図9に示す系を対象とします．これを倒立振子系と呼ぶことにします．

手に相当するのが台車で，それを速度制御系を構成したサーボ・モジュールを利用して直流モータを駆動します．その動きを台車の上にのせた振子に伝えて立たせようとする構成です．座標系は図に示すとおりで，台車の位置をz，振子の角度をθとし，それぞれ右方向，時計回転方向を正とします．

● 状態方程式を立てる

まず，制御対象の特性を表す状態方程式を導出することからはじめなければなりませんが，本章では図9中の記号を利用して非線形微分方程式を導出し，それを線形微分方程式に近似した結果のみを示します(導出の詳細について知りたい方は，たとえば，筆者のWebページを参照．http://feedback.mech.fukui-u.ac.jp/)．なお，ζ，ξはモータ駆動の台車部分の特性を表す物理パラメータです．

$$\begin{cases} \ddot{z}+\zeta\dot{z}=\xi u \\ mL\ddot{z}+\left(J+mL^2\right)\ddot{\theta}-mgL\theta+\mu_\theta\dot{\theta}=0 \end{cases} \quad \cdots\cdots(24)$$

いくつか注目してもらいたい点があります．まず，台車の上で振子が回転運動するので，本当であれば，台車の運動を表現している第1式中に$\dot{\theta}$，あるいは$\ddot{\theta}$に関する項が登場してしかるべきですが，ここで前提としているモータ駆動方式からそのような項は生じません．それから，振子の回転運動を表現している第2式中に$\ddot{z}$の項がありますが，これを通して振子が回転トルクを受けます．

一方，第3項が重力により振子を倒そうとするトルクを表しています．したがって，操作量uを適切に与えることで台車を駆動し，そこで発生する加速度$\ddot{z}$を利用して振子にトルクを与え，重力によるトルクをいかに補償するかが安定に制御する鍵となります．

上式に対して，状態量xを次式のように定めます．

$$x=\begin{bmatrix} z \\ \theta \\ \dot{z} \\ \dot{\theta} \end{bmatrix}$$

このとき，状態方程式は次式で与えられます．

$$\dot{x}=\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -\zeta & 0 \\ 0 & p_1 g & p_1\zeta & -p_2 \end{bmatrix}x+\begin{bmatrix} 0 \\ 0 \\ \xi \\ -p_1\xi \end{bmatrix}u \quad \cdots\cdots(25)$$

ここで，

$$p_1=\frac{mL}{J+mL^2},\ p_2=\frac{\mu_\theta}{J+mL^2}$$

倒立振子系は不安定な制御対象です．しかし，可制御であることを示すことができるので，状態フィードバック制御により

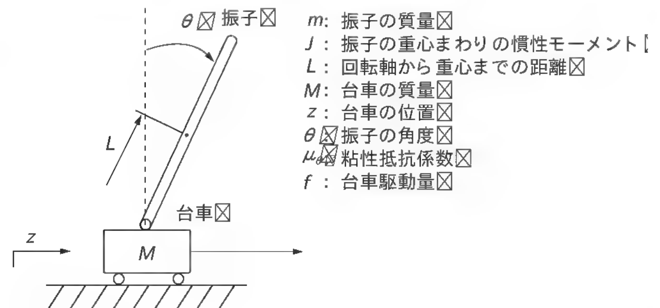


図9　倒立振子系

安定化が可能です．そのためには，状態量xが必要となります．台車の位置zと振子の角度θはポテンショ・メータなどの角度センサを利用することで簡単に直接計測が可能です．なお，回転軸まわりの静摩擦は可能な限り低減したいので，非接触型のセンサを使用します．計測範囲ですが，安定化のみをめざすのであれば，±45°程度あれば十分ですが，振り上げ制御を考える場合，360°計測可能なものを選定する必要があります．

一方，前述の振子系のところでも述べたように，(角)速度については適切なセンサがないために，全状態観測器を利用して推定することを考えます．

● 観測方程式と可観測性を考える

台車の位置zと振子の角度θが直接計測できるとしたときの観測方程式は次式で与えられます．

$$y=\begin{bmatrix} z \\ \theta \end{bmatrix}=\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}x \quad \cdots\cdots(26)$$

このときの可観測行列Wは$n=4$であることから，

$$W=\begin{bmatrix} C \\ CA \\ CA^2 \\ CA^3 \end{bmatrix}=\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{bmatrix}$$

で与えられます．なお，Wの上半分で$rank(W)=4$であることが示されるので，＊の内容は省略してあります．したがって，台車の位置zと振子の角度θが計測できれば，全状態観測器により状態量xを推定できることがわかります．

以上の解析から，台車の位置zと振子の角度θを計測できればよいことは示せましたが，これらをともに計測しなければならないのでしょうか．

このことを確認するために，台車の位置のみ計測できる($y=[1\ 0\ 0\ 0]x$)，あるいは振子の角度のみ計測できる($y=[0\ 1\ 0\ 0]x$)ものとして可観測行列のランクを調べてみます．

(a)台車の位置のみ計測：$rank(W)=2$

(b)振子の角度のみ計測：$rank(W)=3$

表1　倒立振子系の物理パラメータ値

m	0.023	振子の質量［kg］
J	3.20×10^{-4}	重心回りの慣性モーメント［kgm^2］
L	0.2	重心までの距離［m］
μ_θ	2.74×10^{-5}	粘性抵抗係数［Ns/m］
ζ	240	台車系の物理定数
ξ	90	台車系の物理定数

残念ながらいずれの場合も $rank(W) = 4$ を満たしません．これより，倒立振子系においては，台車の位置 z と振子の角度 θ を計測する必要があることがわかります．

● 制御器の設計例とシミュレーション結果

それでは，実際に式(23)に示す制御器の設計を行い，その(初期値)応答を数値シミュレーションにより調べてみましょう．使用した物理パラメータ値を表1にまとめておきます．

このとき状態空間モデルは次式で与えられます．

$$\begin{cases} \dot{x} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -240 & 0 \\ 0 & 36.39 & 890.3 & -0.0221 \end{bmatrix} x + \begin{bmatrix} 0 \\ 0 \\ 90 \\ -333.9 \end{bmatrix} u \\ y = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} x \end{cases} \quad \cdots (27)$$

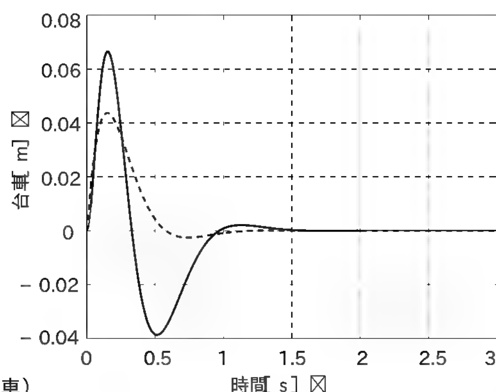


図10
初期値応答（台車）

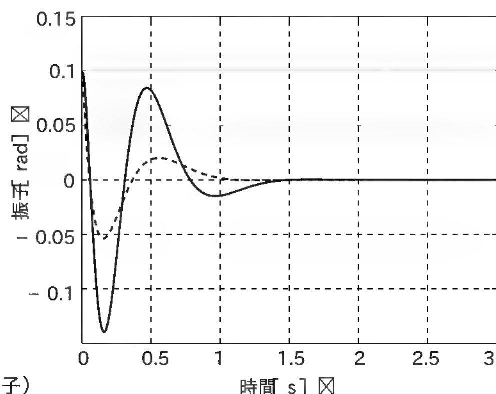


図11
初期値応答（振子）

この状態空間モデルに対して制御器の設計を行いますが，K と H の設計はいずれも極配置法により設計します．指定した極の位置は下記のとおりです．

$$K: \{-240,\ -5+5j,\ -5+5j,\ -7\}$$
$$\rightarrow K = [-25.65\ \ -19.66\ \ -11.58\ \ -3.173]$$
$$H: \{-250,\ -10+5j,\ -10-5j,\ -12\}$$
$$\rightarrow H = \begin{bmatrix} 12.70 & -2.657 \\ -1.224 & 29.28 \\ -156.5 & 554.4 \\ 594.7 & -1892. \end{bmatrix}$$

得られた制御器を次式に示します．

$$\begin{cases} \dot{\hat{x}} = \begin{bmatrix} -12.70 & 2.657 & 1 & 0 \\ 1.224 & -29.28 & 0 & 1 \\ 2465. & 1215. & 802.4 & 285.6 \\ -9157. & -4634. & -2977. & -1059. \end{bmatrix} \hat{x} \\ \quad + \begin{bmatrix} 12.70 & -2.657 \\ -1.224 & 29.28 \\ -156.5 & 554.4 \\ 594.7 & -1892. \end{bmatrix} y \\ u = [\ 25.65\ \ 19.66\ \ 11.58\ \ 3.173\ ] \hat{x} \end{cases} \quad \cdots\cdots (28)$$

初期条件を $z(0) = 0$, $\theta(0) = 0.1$［rad］, $\dot{z}(0) = 0$, $\dot{\theta}(0) = 0$ としたときの応答を図10，図11に示します．なお，図中，点線で描いたのが，状態量がすべて手にできるとして状態フィードバック制御を施した制御結果です．

また，このときの振子の角度 θ に関する推定誤差 $e = \theta - \hat{\theta}$ を描いたのが図12です．なお，制御器の初期状態は $\hat{x}(0) = 0$ としました．

推定誤差が時間の経過とともに0に向かっていることがわかります．ただ，制御を開始してからすぐに推定誤差が0になるわけではありませんので，その影響を受けて，初期値応答が状態フィードバック制御と比較して若干悪くなっていることがわかります．

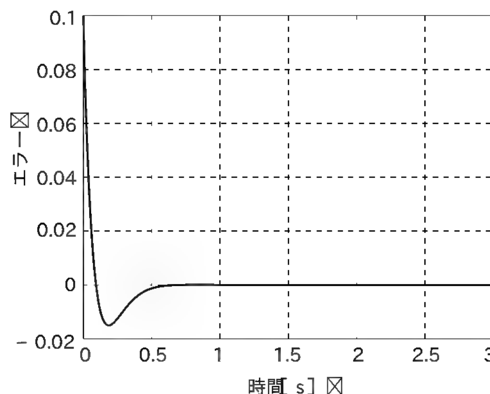


図12
推定誤差

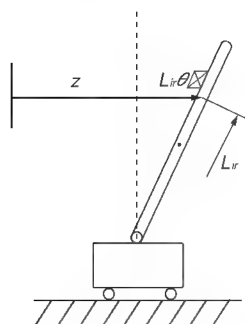


図13 イメージ・センサを利用した倒立振子系の制御

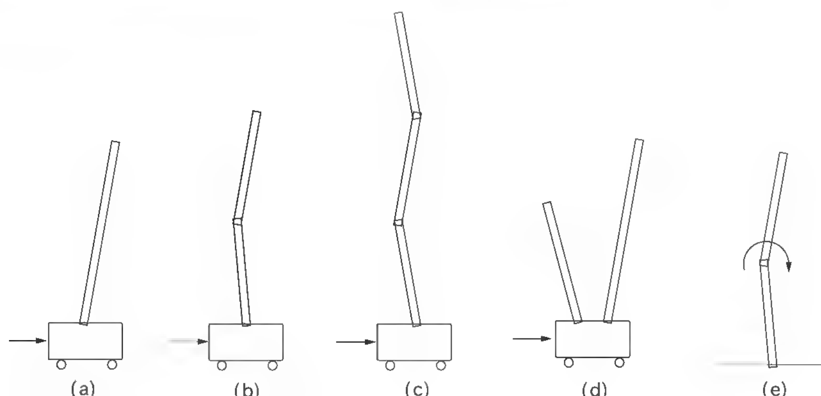


図14 さまざまな倒立振子系

● 画像情報の活用

ところで，倒立振子系を制御するために利用できるセンサは角度センサだけではありません．本誌2001年2月号にCMOSイメージ・センサを利用した倒立振子系の安定化制御についてまとめました．そこでは，台車と振子に赤外線マーカを置き，イメージ・センサを利用してそれらの位置から台車の位置と振子の角度を算出し，制御に利用しました．しかし，イメージ・センサを利用すれば，図13に示すように，あるところを基準点にとり，そこから振子上に設置した赤外線マーカまでの水平距離を計測することが可能となります．

いま，振子の回転軸から赤外線マーカまでの距離をL_{ir}とします．振子の角度θが微小であると仮定すると，観測量は，

$$y = z + L_{ir}\theta = [1 \;\; L_{ir} \;\; 0 \;\; 0]\, x$$

で与えられます．この観測方程式に対して可観測性を調べると，おおよそですがL_{ir}が振子の全長の2/3程度でなければ可観測であることを示すことができます．つまり，この場合は，1個のセンサからの情報ですべての状態量を推定できるのです．

ところで，可観測性という立場からは，それが失われる地点以外であれば，どこに赤外線マーカを置いても状態量の推定は可能です．しかし，制御性能を考えた場合，設置点はやはり重要な問題です．実は，これにはシステムの零点が関与しています．結果からいうと，振子の上部にマーカを置いたほうがよいことがいえます．皆さんが棒を手の上で立てようとするとき，おそらく棒の上部の動きを見ながらフィードバック制御を行っていると思いますが，それに関連した結果であると考えられます．

● より複雑な倒立振子系の場合は？

倒立振子系は，振子の組み合わせ方をいろいろ変えることができる点に一つの特徴があります．図14に，筆者がこれまで扱ったことのある倒立振子系をまとめました．

安定化制御を目的とする場合，倒立振子系に対する(非線形)微分方程式は線形微分方程式で近似したものが使用されます．その場合，台車＋振子の本数だけの連立する2階線形微分方程式が得られます．並列型振子系図14(d)の場合，振子の長さによっては可制御性が失われることが起こりえますが，直列型振子系図14(b)，図14(c)では，理論的には振子の長さや本数に関係なく可制御性を満たします．つまり，状態フィードバック制御により安定化が可能です．センサに関しては，各振子の回転軸に角度センサを取り付けることで可観測性は満たされます．したがって，理論的には式(23)の制御器で安定化制御が可能です．ただし，K，Hの選定は振子の本数や配置に対応して難しくなります．

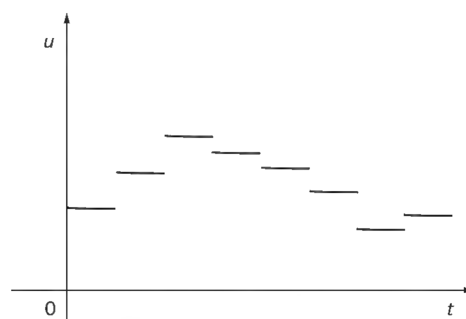


図15 制御器が出力する操作量

制御用プログラム

最後に，設計した制御器のマイコンへの実装について説明します．極配置法や最適レギュレータ法などを利用することで，式(23)に示す制御器を設計できます．この制御器は，線形微分方程式で与えられますが，それを差分方程式で近似する(これを離散化という)必要があります．なぜならば，マイコンではA-D，D-A変換や制御演算に有限な時間が必要であり，マイコンが出力できる操作量は，図15に示すような階段状のものになるためです．

差分方程式は，決められた時間間隔で更新される量を表現するのに適しています．離散化についての詳細は本章ではふれませんが，離散化の一つの手法である零次ホールド法を使用する

ことで次式に示す離散時間制御器を得ることができます．

$$\begin{cases} z[k+1] = A_d z[k] + B_d y[k] \\ u[k] \quad = C_d z[k] \end{cases} \cdots\cdots\cdots\cdots (29)$$

ここで，$z[k]$はサンプリング時間をΔとするとき$z(k\Delta)$を意味しています．つまり，k番目のサンプリング点における z を表しています．式(28)で与えられた制御器をサンプリング時間 0.005[s]で離散化して得られた結果を次式に示します．

$$\begin{cases} z[k+1] = \begin{bmatrix} 1.085 & 0.007556 & 0.01222 & 0.002504 \\ -0.07916 & 1.077 & -0.02656 & -0.004158 \\ 6.128 & 7.423 & 3.213 & 0.8024 \\ -22.69 & -26.48 & -8.218 & -1.977 \end{bmatrix} \\ \qquad\qquad z[k] + \begin{bmatrix} 0.06469 & -0.008094 \\ -0.003186 & 0.1349 \\ -0.1845 & 2.497 \\ 0.756 & -8.366 \end{bmatrix} y[k] \\ u[k] \quad = [\, 25.65 \quad 19.66 \quad 11.58 \quad 3.173 \,] z[k] \end{cases}$$

$$\cdots\cdots\cdots\cdots (30)$$

この離散時間制御器をマイコン(H8/3048F)に実装するためのプログラム例をリスト1に示します．なお，一定のサンプリング時間はマイコンのもつタイマ割り込み機能を利用して実現しています．制御に関する処理は割り込みによって実行される関数内で行われるので，ここではその関数のリストだけを掲載します．

プログラムの内容はすぐに理解できると思いますが，簡単に説明しておきます．

● 013～020行

H8/3048F に内蔵されている10ビットのA-D変換器を利用して，角度センサ(ポテンショ・メータ)から出力される電圧を利用して台車の位置と振子の角度を計測します．なお，振子が鉛直上方に立っている($\theta = 0$)ときにポテンショ・メータから出力される電圧が正確にはわからない，という問題が起こります．そこで，ここでは制御を開始した瞬間には，振子は鉛直上方となるように手で支えられているものとして，そのときの出力電圧を基準と考えます．台車についても同様です．そのために，19行目と20行目で基準電圧分を引き算している点に注意

リスト1　離散時間制御器のプログラム例

```
001 void imia0(void){   /* 割り込み処理用関数 */
002
003    unsigned int d1,d2;
004
005    ITU0.TIER.BIT.IMIEA = 0;   /* 割り込み不許可 */
006    ITU0.TSR.BIT.IMFA = 0;     /* GRA によるコンペア・マッチの発生を示す */
007                               /* ステータス・レジスタ・フラグのクリア     */
008
009    if(flag_exit == 1){  /* 出力制御用フラグがセットされていれば制御器の演算 */
010       DA.DR0 = (int)((u1+2.5)/5.*255.);   /* 操作量の出力 */
011                         /* ±2.5[V] を 0～5[V] へ */
012                         /* D/A 変換器は 8ビット */
013       d1=AD.DRD/64; d2=AD.DRC/64;
014                         /* A/D 変換器は 10ビット */
015                         /* A/D 変換結果は 16ビット の上位 10ビット に置かれる */
016                         /* 64 での割り算は下位 10bit に移動するため */
017       data0 = ((float)d1/1024.*5.);        /* 台車の位置 [V] */
018       data1 = ((float)d2/1024.*5.);        /* 振子の角度 [V] */
019       y1 = (data0-y10)*VOLT_TO_DISTANCE;   /* 台車の位置の計測 [m] */
020       y2 = (data1-y20)*VOLT_TO_ANGLE;      /* 振子の角度の計測 [rad] */
021
022 /* 状態量の更新 */
023       x[0]=xk[0]; x[1]=xk[1]; x[2]=xk[2]; x[3]=xk[3];
024
025 /* 差分方程式 */
026       xk[0]=1.085*x[0]+0.007556*x[1]+0.01222*x[2]+0.002504*x[3]+0.06469*y1-0.008094*y2;
027       xk[1]=-0.07916*x[0]+1.0765*x[1]-0.02656*x[2]-0.004158*x[3]-0.003186*y1+0.1349*y2;
028       xk[2]=6.128*x[0]+7.423*x[1]+3.213*x[2]+0.8024*x[4]-0.1845*y1+2.497*y2;
029       xk[3]=-22.69*x[0]-26.48*x[1]-8.218*x[2]-1.977*x[4]+0.756*y1-8.366*y2;
030
031 /* 操作量の計算 */
032       u1=25.65*xk[0]+19.66*xk[1]+11.58*xk[2]+3.173*xk[3];
033
034 /* リミッタ(安全対策) */
035       if((y1 >= 0.18) || (y1 <= -0.18)){   /* 台車の移動距離 ±18[cm] */
036          flag_exit = 0;   /* オーバしたらフラグをリセット */
037       }
038       if((u1 > 3.0) || (u1 < -3.0)){        /* 操作量の範囲 ±3[V] */
039          flag_exit = 0;   /* オーバしたらフラグをリセット */
040       }
041    }else{   /* 出力制御フラグがリセット(flag_exit=0)されている場合 */
042       u1 = 0.0;
043       DA.DR0 = (int)((u1+2.5)/5.*255.);    /* 0[V] の出力 */
044    }
045    ITU0.TIER.BIT.IMIEA = 1;   /* 割り込み許可 */
046 }
```

してください．

● 022～032 行

差分方程式を解き，操作量の算出を行っています．なお，操作量は $u[k] = C_d x[k]$ により計算しますが，プログラムでは，それを $u[k+1] = C_d x[k+1]$ として計算している点に注意してください．そのために操作量の計算には xk を使用し，得られた操作量は，次の割り込み処理の最初に出力する（010行目）ようにプログラミングされています．

また，差分方程式を解く際に，行列演算を行わなければなりません．for 文を利用することもできますが，演算時間を短くするために，ここでは for 文は使用していません．

● 035～040 行

ソフト的な安全対策です．振子の角度が大きく傾いたとき，あるいは操作量が過大になったときに，フラグ flag_exit をリセットして出力を停止します．

● 10，43 行

H8/3048F に内蔵されている 8ビットの D-A 変換器を利用して操作量を出力しています．ただし，台車は左右方向に動かさなければならないので，D-A 変換器の出力を OP アンプを利用して 0～5[V]から±2.5[V]にシフトしています．そのために，2.5を足し算しています．

おわりに

本章では，フィードバック制御を行う立場からセンサについて考えてみました．線形制御理論では状態フィードバック制御が基本となりますが，必ずしも制御対象のもつすべての状態量をセンサを利用して計測できるわけではありません．そこで，制御対象の特性を表現する状態空間モデルを利用して状態観測器を設計し，それにより推定した状態量を制御に活用することが通常行われます．そのためには可観測性を満足しなければなりませんが，それがセンサの選定基準となります．状態観測器を利用すると，センサで直接見ることができないものも推定できます．そこで，一つの事例として，倒立振子系を取り上げました．この場合，角度センサは容易に利用可能ですが，角速度の入手に問題があるので，それらを状態観測器で推定しました．

本章で取り上げたのは，もっとも基本的な全状態観測器のみですが，これ以外にもいろいろな観測器が研究されています．興味ある方はぜひ勉強してください．

かわたに・りょうじ　福井大学工学部機械工学科
http://feedback.mech.fukui-u.ac.jp/

組み込みソフトウェア業界でどう生きるか

～経済産業省による調査と施策から探る～

【猪飼 國夫】

情報機器の最先端の分野では，30年以上前から多くの技術者が組み込みソフトウェアの仕事に携わってきましたが，行政や学問の分野ではあまり重要視されていませんでした．そのような中，ここ数年来，東京大学の飯塚悦功教授を中心としたSESSAMEプロジェクトでは人材の育成を行い，企業人の門田浩氏を中心にスタートした組み込みネットでは最新情報の共有という形で，危機をいち早く汲み取って活動を続けています．

ソフトウェア業界を管理する経済産業省もここ数年，この分野の重要性に注意を傾けるようになり，去る6月には(独)情報処理振興事業団(IPA)の主催で,「組込みソフトウェア開発力強化推進フォーラム」が初めて開催され，詳細な調査結果と今後の施策などが発表されました．

これらの動きを整理して，フォーラムで発表された経済産業省による調査結果と施策から，この仕事に従事している技術者の将来を予測してみたいと思います．

1 組み込みソフトウェアの業界の危機

● 組み込みソフトウェアの世界

組み込みソフトウェアは，それ専門の技術者というよりは，表1に示すようにいろいろな分野の技術者達によって作成されてきました．

表1[1] 組み込みソフトウェア技術者の内訳

ソフトウェア技術者	41.2%
システム技術者	25.8%
テスト/品質管理技術者	12.3%
管理者/専門技術者	10.3%
その他の技術者	9.5%

当然ながら，それに対しての学問的な裏付けや業界としての標準化への取り組み，技術そのものの評価や認定などはほとんどなされないままであり，企業間競争の狭間で個々の技術とそれに携わった技術者達は使い捨てにされてきたきらいがあります．

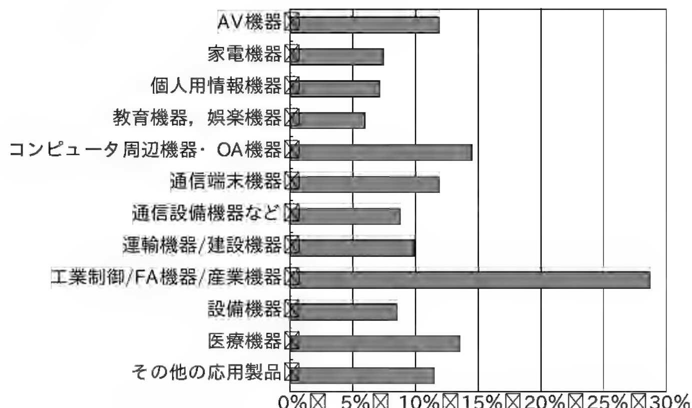


図1[1] 開発している主要な製品・サービス(複数回答可)

最近になって組み込みソフトウェアへの関心が高まるとともに，いろいろな問題点が指摘されるようになりました．これまで組み込みソフトウェアに力を割いていなかったいくつかの雑誌でも，前記フォーラムで発表されたデータを参考にして，最近の組み込みソフトウェアの不具合に起因する製品の回収事故と経済産業省の施策をトレンドとして紹介したり，組み込みソフトウェアの啓発記事を特集として取り上げています[2],[3]．

ソフトウェアの危機の芽は，組み込みソフトウェアが4ビット・マイコンで使われ始めた当初からあったのですが，この頃は開発技術者にハードウェア出身の人も多く，持ち前の器用さで危機が表沙汰になることなく処理されてきました．

組み込みソフトウェアの危機があまり指摘されてこなかった原因は，図1に示すように組み込みソフトウェアを必要としている製品やサービスが多岐にわたっていて，声が一つにまとまらなかったからです．それだけでなく，組み込みソフトウェアについて今後のことを考えると，機器やサービスの表に出る部分とリアルタイム・サービスなどの基本的な部分，その中間の抽象化できる処理部分などに切り分けて作成するという共通認識を業界や教育界でもつ必要があります．そうしないと，それぞれに特化した技術者を育成したり，技術の標準化・汎用化や保存に適さないということです．

● どのような危機があるのか

いま騒がれている組み込みソフトウェアの危機とは，次のような状況によるものです．

1) ソフトウェアの規模が大きくなり，個人や小人数のグループではとうてい開発できなくなり，多くの人が開発に参画することによって全体の把握が困難になった
2) 組み込みソフトウェアが機器単体で動く時代から，複数の機器や機能の連動を必要とするようになり，開発者がそれぞれ個別の概念で開発していたのでは，システム全体を統一的に動かすことが難しくなった
3) 組み込みソフトウェアが機器などの性能を大きく左右するようになり，この分野での覇権を握った者に大きな利益をもたらす可能性が高くなった
4) 組み込みソフトウェアを開発する人材を育成，供給，評価する構造が成り立っていないため，個々の職場単位で新卒者から実践要員を仕立てあげており，知識や意識の基盤がばらばらで，人材の幅広い活用と効率的な利用ができない
5) 日本以外の国でも組み込みソフトウェアの分野に進出する

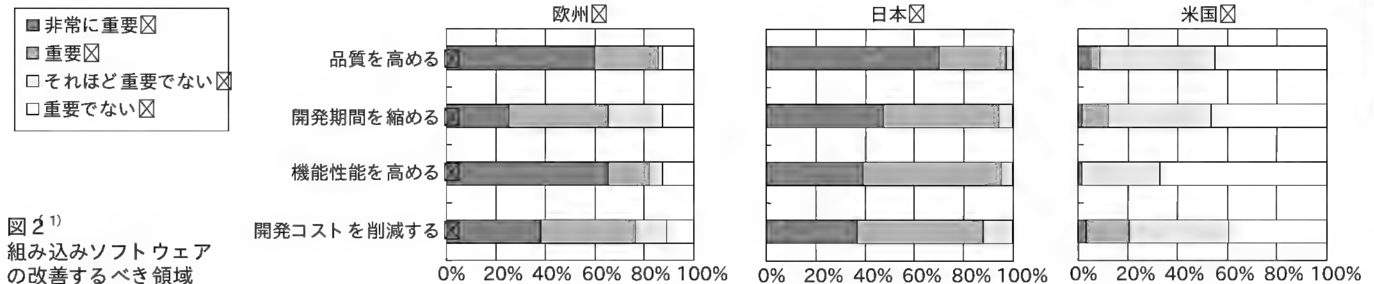


図2[1)]
組み込みソフトウェアの改善するべき領域

国や企業が増えて，独自のノウハウや技術基盤そのものの流出が起こっている

● 危機の認識

飯塚教授の認識によれば，このような事態を招いた大きな原因として，日本の行政，個別企業，教育界の戦略性の欠如を挙げています．以前にもどこかで聴いたことがあるような議論です．日本語での処理に寄りかかったワープロなどのビジネス・ソフトウェアやメモリICにぶら下がった半導体業界などは，完全に成り立たなくなってしまいました．それは，日本がその当時有していた有利な条件に依存して，ただ戦術的な観点からだけで，そこに資源を傾注してきたことから起きた危機です．

使い方に問題がないわけではない外国製のワープロの席捲を防ぐことも，日本発のオープンなOSであるBTRONの普及も，農業生産物の保護のような選挙の票目当ての戦略なき政治的妥協により，功を奏するに至りませんでした．

日本が現時点でかろうじて世界に発信できているソフトウェア分野は，ゲームと組み込みソフトウェアでしょう．しかし，こちらも漫画文化を背景としたゲームやゲーム機，および家電や制御用の組み込み機器が日本でたくさん開発・生産されてきたという，特殊な事情が背景にあるだけで，特別にそうなるような政策や戦略を打ち出した結果ではありません．

日本風の漫画やゲームが，アジア諸国だけでなく世界中で歓迎されている状況下では，そう遠くない将来にそれらの開発に携わりたいという，多くの優れた若者が世界中で輩出されると思われます．すでに日本のアニメの画面の制作はアジア諸国に依存しています．家電や制御機器の製造拠点が中国などに転移するにつれて，それらの国で組み込みソフトウェアが開発されるのは当然のことでしょう．

2 調査結果に見る組み込みソフトウェアの現状と問題点

経済産業省の調査報告書から，いま各企業が組み込みソフトウェアに対して抱いている認識をいくつか選び出してみます．

● 企業は何を最重要視しているか

図2からわかるように，日本企業は欧州企業とよく似た考えをしていますが，米国の企業は組み込みソフトウェアに熱心でないせいか，開発コストの削減以外には関心が低いようです．

開発費に占める組み込みソフトウェアの割合は，日本では半数以上の企業が1/3程度以下で，半分以上を占める企業も10%近くあります．これに対して米国では60%以上を占めることはほとんどなく，欧州では60%以上は0という状況です．

表2[1)]
平均的な組み込みソフト開発期間（%）

地　域	欧州	日本	米国
2年以上	0	1.5	1.7
1.5～2年	8	3	9
1～1.5年	8	10	10
0.5～1年	40	41	35
0.5年以下	33	40	43
不明	8	12	7

ソフトウェアの開発期間は表2のように，1年以下の短期間で開発することが要求されています．

● 外部/海外委託

日本企業は組み込みソフトウェアに限らず，多くの業務を子会社や零細・中小企業に下請・外注することで，自社の高コスト体質をカバーしてきました．外部に委託している企業は38%，子会社など関連会社への委託を含めると72%の企業が自社だけでは人員が足らないということを最大の理由（45%）として，外注や下請けを使っています（図3）．それは当然ながら海外の受託先の利用にもつながっています（図4）．

この調査は政府による任意調査である以上，これらの調査結果から各企業の戦略を読み取ることは難しいと思われます．しかし，企業各社があまり確たる戦略がないまま，自社の当面の利益を確保するという姿勢であることは読み取れそうです．

3 海外との競争はどうなるか

組み込みソフトウェアが日本で発達してきたからといっても，それだけでずっと日本が優位に立っていることはできません．

● 中国の脅威

中国でIT産業が集まっている地域は，先月号と今月号に掲載されている「電脳事情にし・ひがし」で触れているように，北から大連市，北京市，長江河口域，珠江デルタなど南北2,300kmにまたがっています．

大連市は日本企業が，北京市はMicrosoft社をはじめとして米国企業が，ソフトウェアを中心にして進出しています．長江河口域や珠江デルタ地帯は巨大産業地域で，ICなどを含むハードウェア産業が多く集まっています．

中国は現在，国内各地で建設されている飛行場や高速道路，産業基地，ホテルや事務所ビルなど多くの例で示されているよ

[redacted]

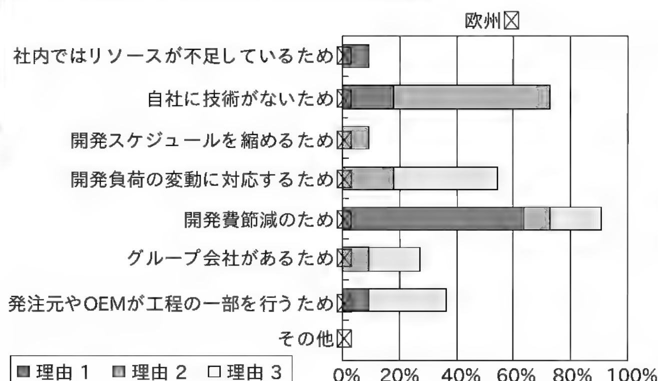


図3[1] 外部リソースを使う理由（理由1が最大理由，理由2は2番目，理由3は3番目の理由）

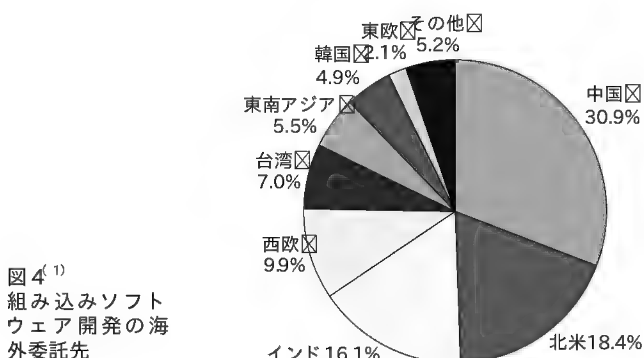


図4[1]
組み込みソフトウェア開発の海外委託先

うに，お互いの見栄の張り合いからか，外国から見ると無謀とも思えるような大きなシステムを作り上げることがあります．計画性と品質に問題があってむだになることも多くありますが，ある日，何かの要因でそのシステムがフル稼働するときがあると，この遊んでいたシステムがすでに用意されていることで，迅速な立ち上がりが可能になり，恐るべき速さで新しい分野へ進出してしまいます．これは三千年の歴史が物語る真実です．

組み込みソフトウェアの世界でも，たった一人の天才が出現すると，ITRONのように日本では普及にたいへんな努力を要したものでも，一気呵成に全世界の基準を変えてしまう可能性は否めません．たとえば，別稿（電脳事情にし・ひがし）で紹介している中国最大の独立系IT企業東軟集団も，瀋陽市の東北工学院（現東北大学）の劉博士が1991年に同学院のコンピュータ・ネットワーク工学研究室を主体に創業し，2003年には6,000人の従業員で300億円近い売り上げを出しています．労働集約型産業なので，この数字は中国の物価水準からみると，日本の1,000億円以上の売り上げに相当します．

● シリコンバレーの現状

この地域は世界有数の情報産業の集積地ですが，本誌の過去の連載からも伺われるように，家賃の高騰や子供の教育など，集中の弊害が出てきており，ある意味では日本と同じような悩みを抱えています．

しかし，本家本元の強みは世界中から優秀な人材が集まってくることにあります．新しい仕事がつねに創設され，それとともに起業したところに雇用が発生します．

ただ，組み込みソフトウェアの分野に関しては，ここに集まってくる一旗上げたい技術者たちにとって大きな関心事ではないようです．組み込みソフトウェアは，どうしても組み込まれる機器の存在があるので，家電・自動車や制御機器などが開発・製造されている拠点や企業との関係が重要です．

伝統的に日本式の下請け構造を持たない欧米の企業では，このような開発は勢い社内で処理することが多かったため，独立系の組み込みソフトウェア会社が立ち並ぶという状況はあまりありません（図5）．独立系の企業は，独自の制御用のソフトウェアを作成して，機器を開発している企業に売り込むという形での参入が多いようです．

● EMS企業の台頭

米国のような状況では，日本型の組み込みソフトウェアの制作企業は，うまくやれば共存を図ることも可能かと思われます．しかし，米国では電子機器の製造に関してアウトソーシングを受けるEMS（Electronics Manufacturing Services）企業が台頭していて，この動向でいつ日本型組み込みソフトウェアの世界が変わるか，予断を許しません．

携帯電話機製造などのEMS企業は機器の設計などの上流分野と販売などの下流分野の中間の，もっとも利益が出にくい組み立て製造に特化した企業ですが，スマイル現象といわれるこの両端が上がった利益構造のどちらかに進出しようと機会をうかがっていると考えたほうがよいようです．中国においてはEMS企業はすでに軸足を大きく企画・開発へとシフトしています．

その流れが主流となった場合，図6のように小さなグループが主体の日本の組み込みソフトウェアの開発体制は，根底から崩れ去るでしょう．

4 日本の現状

● 意図的努力なしで組み込み分野で地歩を占めていた日本

いままで日本の組み込みソフトウェアが世界的に有力であり続けてきた理由の一つに，組み込まれる機器類を日本国内で開発してきたという事情があります．自動車やカーナビ，携帯電

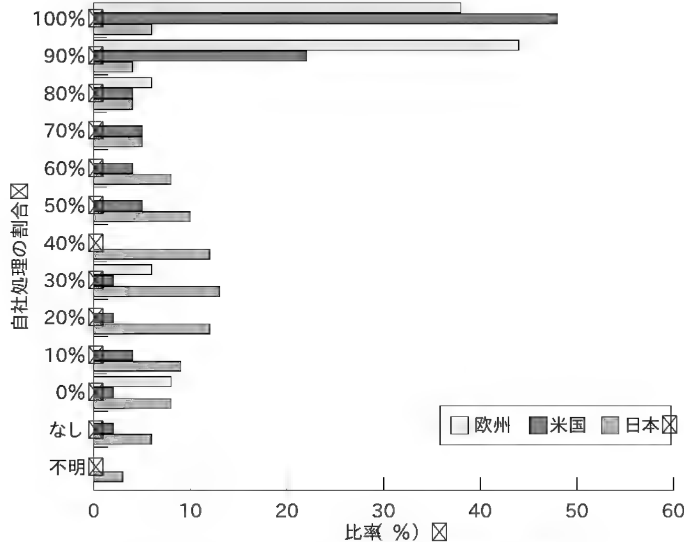


図5[1] 日本と外国でのソフトウェア設計の自社開発の比率

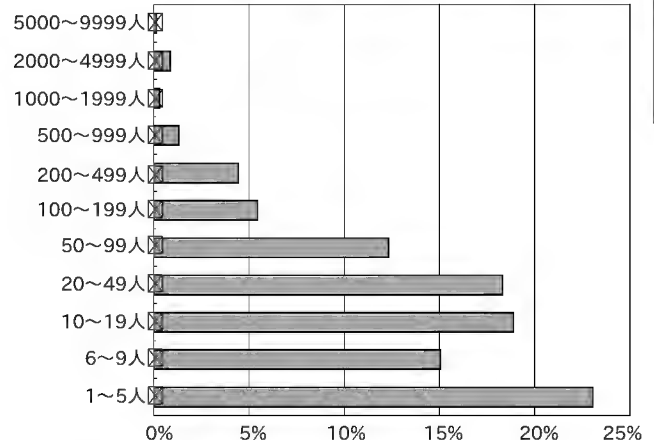


図6[1] 組み込みソフトウェア技術者の人数分布

話，種々の白物家電製品，AV機器，ロボットをはじめとするFA機器や制御装置などすべて日本国内で，日本人の手で開発されてきました．これは，経営戦略や政治目標でそうなったのはなく，医師などのように社会的な評価が高いわけではない，オタク的な技術者が個人的なスキルとしてそのノウハウをため込んできた結果です．

● 内・外部環境の変化

この開発の体制にこの10年来大きな変化が起こってきました．一つは優秀な開発技術者の不足です．大学などの高等教育機関からは多くの人材が供給されてきていますが，最優秀な人材は1970年代からだんだん工業・工学分野に進学しなくなってきました．より有利な人生が送れそうな医学や国際関係に進む人が増えたためです．

もう一つは開発時間の短縮です．商品のライフ・サイクルがどんどん短くなり，パソコンや携帯電話に至っては半年前の最新機種は安売りの対象になってしまいます．それに対処するためには，数多くの質がそろった技術者を大量に集めて，仕事が終わった後は遊休人員にならないようにする必要があります．

このような事態に対応するには，以前は下請け企業を使っていましたが，下請企業の管理費が高くつくため，派遣技術者を使うのがいちばん安上がりになりました．いま日本では，技術者を派遣する企業が隆盛をきわめていて，名が通っている大学の卒業生以外は，たとえ親会社として大企業の看板を掲げていても事実上派遣社員にならざるを得ない事態になっています．

● 技術基盤喪失の危険性

すでに日本が開発から製造までを放棄したIT関係の機器にはパソコン本体があります．筐体，CPU，メモリ，チップセット，マザーボード，BIOS，グラフィックLSIおよびカードなどほとんどがアメリカや台湾，中国で開発・生産されたものです．

白物家電製品は比較的価格が安いものをはじめとして，製造だけでなく開発までもが国外で行われつつあります．大連の例のようにカーナビなどの高度な組み込みソフトウェアも，日本語ができる人材さえいれば，より安価なところへ移動しつつあります．

下請け依存状態でもそうでしたが，開発を派遣社員と国外に依存したのでは，ますます若い技術者を育て上げる技術基盤が弱くなってしまいます．日本の状況がよかったときに，人材育成のシステム構築に投資をしておかなかった失敗が裏目に出ているのです．小泉総理が主張した米百俵の考えは，思いつきではなく本当に実行してこそ意味があるのです．

5 技術者にとっての危機管理

現状の組み込みソフトウェアで発生している問題点の細かいところは，経済産業省の調査報告書などを参照していただくとして，最後に技術者の側から見たこの状況への対処の方法を考えてみます．

1) 圧倒的な実力を付けて，どこの企業でも渡り歩ける一匹狼エンジニアとなる(多分に米国の技術者的な生き方となるので，外資系が向いているかもしれない)
2) 英語か中国語を完全にマスタして，日本企業にかじりつかない(日本人に特有な発想だけではやっていけないから，人の考えを理解する習慣を身につける必要がある)
3) 別のもっと楽で給料が保証されている公務員などの職に転向する(いちばん難しい選択かもしれない)

など，どれを取っても太平楽ではないようです．

とにかく，英語で仕事ができる程度の実力がないと，10年後にこの業界で生き残っていることは難しいかもしれません．

参考・引用*文献

(1)* 経済産業省,「2004年版組込みソフトウェア産業実態調査報告書」(http://www.meti.go.jp/policy/it_policy/technology/technology.htm)
(2) 日経エレクトロニクス誌2004年7月5日号「デジタル家電のソフト開発不具合ゼロへの初めの一歩 – 高まる開発プロセス見直しへの気運– 」
(3) 情報処理，2004年7月号特集「組み込みソフトウェア開発技術」

いかい・くにお　工学博士,(株)エム・アイ・ベンチャー

組み込みプログラミング・ノウハウ入門(第19回)

ノンブロッキング・プロトコルとロック・フリー・プロトコル

――タスク間のデータ共有の効率を追求する

藤倉 俊幸

1 はじめに

● マルチタスク・システムの性能を左右する要素とは?

2回続けてRTOSなしでも使えるテクニックについて説明した.今回もその続きでRTOSなしでも使えるデータ共有の説明を行う.この方式はRTOSがあっても使用できる.

マルチタスク・システムを構築する場合,タスク間での通信や同期の効率が良いか否かは非常に重要な問題である.定石としては,セマフォやメール・ボックス,データ・キューなどのRTOSが提供する機能を使用してタスク間通信・同期を行う.しかし残念ながら,これらは必ずしも効率が良いとはいえない場合がある.

たとえば,セマフォを使うとタスクがプライオリティに関係なくブロックされるため,余計なコンテキスト・スイッチが起こって時間効率が悪い.また,メッセージなどのコピーをともなう通信手段を使えばセマフォのような排他制御は不要になるが,コピーする分だけメモリ効率が悪くなるし,コピーする時間すら惜しいということもあるかもしれない.

効率の良い方法とは,排他制御もメモリ・コピーも必要のない方法である.そのような方法があるのだろうか.RTOSを使用しない場合は,並行動作するのは割り込みハンドラとメイン関数だけになる.この場合にデータ共有するには割り込み禁止を使う.もう一歩進んで,割り込み禁止を使わないでデータ共有する方法はないだろうか?

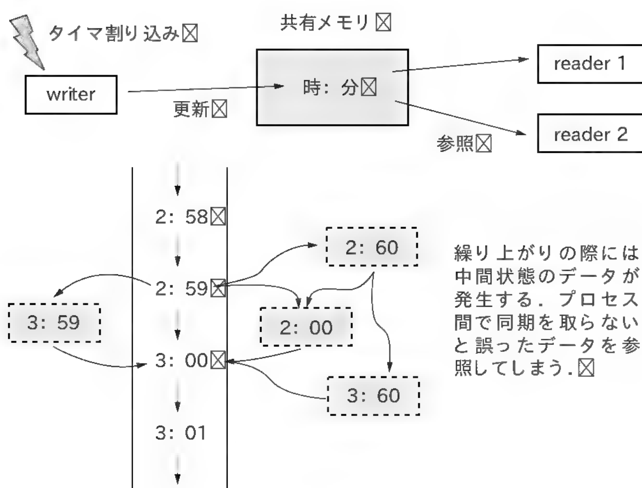


図1 構造を持ったデータの例

● 効率よくデータ共有できる二つの手法

組み込み系でよく使用されるタスク間通信のパターンは,一つのタスクだけがデータを書き換えて,そのデータをほかのいくつかのタスクが参照するパターンである.しかも,参照する側は最新のデータにのみ関心がある場合である.

このようなときに使えるのが次項で紹介する「ノンブロッキング・プロトコル」[1]である.この方法は,もともと複数のコンピュータから構成される分散システム用に提案されたものである.しかし分散システムでなくとも,組み込み系で割り込みハンドラが周期的に更新するデータを複数のタスクが参照するような場合にも使うことができる.

それからもう一つ,タスク間の役割分担が対称な場合,つまり書き手と読み手の役割が分かれていない場合,あるいは書き手が複数いる場合にも使える手法として「ロック・フリー・プロトコル」[2]を第3項で紹介する.これらの手法はリトライ・ベースの手法といわれる.どちらの手法も使い方によっては排他制御を使用するより効率が良いことが報告されている[1][2].

● データ共有のためには同期が必要になる

通信対象となるデータが整数値のような一つのCPU命令で読み書きできる程度のもの,いわゆるC言語の基本型であれば共有メモリにそのデータを配置するだけで(ハードの制約がなければ)排他制御を気にせず自由にアクセスすることができる.しかしデータが何らかの構造を持っている場合には,同期を取る必要が出てくる.

図1に示した例は,一つのタスク(writer)が共有メモリ上の[時:分]形式で表される時刻データを更新し,ほかの二つ(reader1, reader2)がそれを参照しているものである.更新すべきデータに構造がある,つまり時間と分の関係があるため,繰り上がりが発生するときには時刻データが不定になる中間状態が発生する.したがって更新中には,readerがデータを参照しないようにするため,タスク間同期が必要になる.

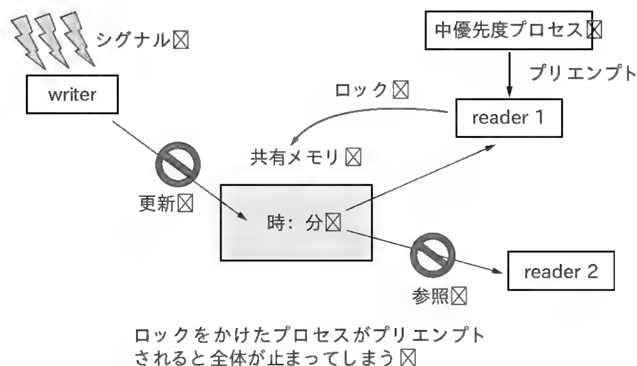


図2　プライオリティ・インバージョン

リスト1　ノンブロッキング・プロトコル

```
初期化:
    CCF = 0;              // 制御変数の初期化

Writer:
    CCF_old = CCF;
    CCF = CCF_old + 1;
    <共有データ書き込み処理>
    CCF = CCF_old + 2;

Reader:
start:  CCF_begin = CCF;
    if (CCF_begin & 1)  // 奇数の場合はやり直し
        goto start;
    <共有データ読み込み処理>
    CCF_end = CCF;
    if (CCF_end != CCF_begin)
        goto start;
```

● 定石はクリティカル・セクションだが…

一つの解決策としては，データを更新する部分と，参照する部分をクリティカル・セクションとして同一のセマフォでブロックする方法が考えられる．これが組み込みシステムにおける定石である．しかし，セマフォを使うとコンテキスト・スイッチが発生する確率が高くなってしまう．たとえば，優先度の高いタスクがクリティカル・セクションに入ろうとしたとき，そこがすでにロックされていると必ずコンテキスト・スイッチが発生し，そのクリティカル・セクションをロックしているタスクにコントロールを移そうとする．時刻を参照するだけなら数十nsオーダ(10^{-9}s)の時間で終了するのに，数十μsオーダ(10^{-7}〜10^{-6}s)のコンテキスト・スイッチが起こったのでは釈然としないものがある．

さらに，参照中はwriterの更新がブロックされるので時計が遅れるなどの影響が出る可能性もある．たとえば，読み込みタスクが共有データをロックして読み込んでいる最中に，別の優先度の高いタスクにプリエンプトされてしまうことが考えらる．このプリエンプトされている時間が長いと時刻データがだんだん狂ってしまう(図2)．このプリエンプトしているタスクがwriterよりも優先度が低い場合は，いわゆるプライオリティ・インバージョンになる．つまり，「readerよりも優先度が高いがwriterよりも優先度の低いタスク」が間接的にwriterをブロックしてしまう現象である．プライオリティ・インバージョンを避けるためには，プライオリティ継承プロトコルなどを使うことになり，次々と高度なシステム・コールが必要となる．

2 ノンブロッキング・プロトコル

● ノンブロッキング・プロトコルとは何か

このような場合に利用できる良い方法が報告されている．ノンブロッキング・プロトコルである．この方法では，書き込み側に優先権を与えていつでも書き込み可能にし，一方で読み込み側は，読み込み中に書き込み側によってデータが更新された場合に再度読み込みを行うというものだ．

セマフォを使う方法をロック・ベースと呼ぶのに対し，ノンブロッキング・プロトコルは再度読み込みを行うことからリトライ・ベースの手法と呼ばれることがある．ロックを使わないため余計なコンテキスト・スイッチが発生しない．この手法では，書き込みが行われたことを示す一つの制御変数を使用する．これだけなのでRTOSとは関係なくアプリケーションのみで実現できる．

疑似プログラミング言語で書いたソース・コード例をリスト1に示す．共通制御変数であるCCFへのアクセスは不可分(アトミックという場合もある)なものでなければならない．つまり，単一のCPU命令で読み書き可能であり，途中で割り込みなどが起きても中断されない必要がある．具体的には，一つのロードかストア命令になるようにint型などを使う．しかし，最近よく使われるRISC型CPUの場合には，単純な型の代入文でも複数のCPU命令で実行されてしまう．この場合でも，レジスタだけを利用した複数のCPU命令であれば普通は問題なく使える．ただし，コンパイラの最適化などで文の順序が入れ替わったりしていると困るのでアセンブラ・コードで確認したほうが良い．

動作はまず，CCFを0に初期化しておく．そして，共通のデータ構造に書き込むタスクは，書き込み中はCCFの値が奇数，書き込み後は偶数になるようにコントロールする．読み込むタスクは，CCFを見て偶数になるまで待機し，偶数になれば必要なデータを読み込む．そして，読み込んだ後CCFが読み込み前と変化していないか確認する．もしCCFの値が変化していれば読み込み中に書き込みが行われたことを示しており，この場合は，読み込んだデータ構造が矛盾している危険性があるのでもう一度最初からやりなおす．

この方法では，書き込みタスクが書き込もうとしたとき，読み込みタスクが読み込み中であってもコンテキスト・スイッチが発生しないので，OSによるオーバヘッドはない．たとえばセマフォを使っていると，

　書き込みタスクが書き込もうとしたときにブロックされて
　→コンテキスト・スイッチが発生

→読み込みタスクが動き出す

→読み込み処理が終了

→クリティカル・セクションを出る

→コンテキスト・スイッチが発生

→ようやく書き込みタスクが新しいデータを書き込む

つまり，わざわざコンテキスト・スイッチを2回もやって古いデータを読み込むのである．また，この方法ではブロックも行われないのでプライオリティ・インバージョンも発生しない．

● リトライは何回発生するか

問題は，readerのリトライ回数であるが，読み込み処理時間d_rと書き込み処理時間d_wはほとんど変わらない条件での最悪の繰り返し回数は，分散システムの場合は，図3に示した場合の3回になる．計算の詳細は省略するが，reader側の最悪実行時間は次の式で計算される．なお，$\lfloor x \rfloor$はxの小数点以下を切り捨てる記号である．

$$3d^{rw}\left\lfloor \frac{\ell_0 + mint - 3d^{rw}}{mint} \right\rfloor$$

d^w　読み込み/書き込み処理時間, ☒$^{w} \cong d_r \cong d^{rw}$

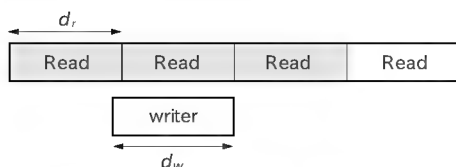
$mint$　書き込み間隔☒

ℓ_0　読み込みタスクの余裕時間☒
（デッドライン－実行時間）☒

単一CPUシステムの場合には並列動作はできないので，基本的にリトライ回数は1回になる．したがって，スケジュール可能かどうかは，1回分のリトライを考慮してレート・モノトニック法などを適用して時間解析を行えばよい．

この手法を単一CPUシステムで使ううえで注意すべき点は，`Reader`タスクは`Writer`タスクよりプライオリティを低くすることである．同一あるいは`Reader`タスクのほうがプライオリティが高いと，`Reader`タスクはスピン・ロック（待機中でもCPU時間を放棄しない）しているので無限ループしてしまう．また，プライオリティ・ベースのプリエンプトな環境でなければならない．つまりITRONをはじめとする，いわゆるRTOS環境が必要である．どうしても`Writer`と`Reader`を同一プライオリティにしなければならない場合には，リスト2のようにするのはどうだろうか．

`Reader2`は，ITRONを想定している．ここではレディ・キューを回転させて，ほかのタスクにCPUを使用する権利を与えている．長く待たされているタスクのプライオリティがだんだん高くなるエイジング機構を持っている場合には，`Reader`タスクのプライオリティの管理に注意する必要がある．



Write時間とRead時間がほぼ等しい場合の最悪リトライ回数は3回☒

図3　readerの最悪リトライ回数

3 ロック・フリー・プロトコル

● 複数のタスクが読み書きしたい場合はどうするか

次にロック・フリー・プロトコルを説明する．ノンブロッキング・プロトコルは，一方が書き込み専門，他方が読み込み専門でタスクの役割分担が決められていた．しかし，ロック・フリー・プロトコルでは，対称的な動作をするタスクに適用することができる．つまり，一つ一つのタスクが共有データを読み込んで，変更して，書き戻す場合に使用することができる．

ただし，使用するためには条件があり，最短連続実行時間，つまり一つのタスクが連続実行できる時間が保証される必要がある．たとえば，ラウンドロビン・スケジューリングのタイム・スライスのような時間でも良い．そして，その時間が共有データを読み込んで書き戻すまでの時間に比べて十分に長い必要がある．つまり，完全なイベント・ドリブンのシステムではだめで，非同期でいつプリエンプトされるのか不明な場合には使えない．

しかし，イベント・ドリブンでも，最小イベント間隔が保証されていれば使用できる．最小イベント間隔を最短連続実行時間と考えれば良い．つまり，いつプリエンプトされるのかわからなくても，一度プリエンプトされたらしばらくはプリエンプトされない期間が保証されていれば使えるのである．

ロック・フリー・プロトコルのコード例はリスト3のようになる．

この関数`enqueue`は，与えられたデータ`input`を待ち行列リストにつなぐものである．一つの待ち行列に複数のタスクからデータを加える場合は，行列をロックして排他制御するのが一般的だが，このプロシージャを使えば複数のタスクから排他制御なしで行列につなぐことができる．`CAS2`は，two-word compare-and-swapルーチンで，第3引き数と第4引き数を比較して同一の場合，第5引き数と第6引き数をそれぞれ第1引き数と第2引き数が指すアドレスに格納するものである．そして，比較の結果が同一であれば`True`を返す．これらの処理を不可分に実行する必要があるため，`CAS2`実行中だけは割り込

リスト2　writerとreaderを同一プライオリティで実装する場合

```
Reader2:
start:
    CCF_begin = CCF;
    if (CCF_begin & 1)        // 奇数の場合はやり直し
    {
        rot_rdq(TPRI_SELF); // CPU放棄
        goto start;
    }
    <共有データ読み込み処理>
    CCF_end = CCF;
    if (CCF_end != CCF_begin)
        goto start;
```

リスト 3　ロック・フリー・プロトコルのコーディング例

```
#define NULL    ((void *)0)
typedef int data_type;

typedef struct Q_Type {
    data_type data;
    struct Q_Type *next;
} q_type;

/* shared variable */
q_type *head, *tail;

/* local variable */
q_type *old, *newData;
q_type **addr;

void enqueue(data_type input)
{
    newData->data = input;
    newData->next = NULL;
    do {
        old = tail;
        if (old != NULL)
            addr = &old->next;
        else
            addr = &head;
    } while(!CAS2(&tail, addr, old, NULL, newData, newData));
}
```

み禁止状態にしなければならない．

● Read-Modify-Write

もっと単純なRead-Modify-Writeの場合のコーディングはリスト 4のようになる．

addrが指す共通領域からデータをoldに読み出して，関数fncで新しい値を計算してまたaddrが指す領域に戻す．(1)と(2)で新しい値を計算する．CASは，addrが指す領域の値とoldが同一であればnewをaddrが指す領域に書き込む．同一でない場合CASは，FALSEを返すがこのときは，(1)から(3)までの間にコンテキスト・スイッチが起きたと判断される．こ

リスト 4　Read-Modify-Write

```
#define FALSE 0
#define TRUE 1
typedef int data_type;

void RMW(data_type *addr, data_type *fnc(data_type))
{
    data_type old, newData;

    old = *addr;                                    …(1)
    newData = *fnc(old);                            …(2)
    if (CAS(addr, old, newData) == FALSE)           …(3)
    {
        old = *addr;                                …(4)
        *addr = *fnc(old);                          …(5)
    }
}
```

の場合は，再度読み出しからやり直すことになる．ここで，前提から最短連続実行時間が保証されていて，しかもそれがRead-Modify-Write処理時間より十分長いことも保証されているので，一度だけリトライすれば十分なのでループする必要はない．

● スケジューリングのようす

図4は，三つの周期タスクT_1，T_2，T_3がロック・フリー・プロトコルで二つの共有領域S_1，S_2を共有しながらレート・モノトニック・スケジューリング法で処理するようすを示したものである．優先順位はT_1，T_2，T_3の順である．共有領域S_1はT_2とT_3が利用し，S_2はT_1とT_2が利用する．

ノンブロッキング・プロトコルでは書き込み側が優先されるが，このロック・フリー・プロトコルではそれぞれのタスクの優先順位に従って処理が進む．図4では，優先順位の低いタスクほどリトライ発生回数(グレーの部分)が多くなっていることがわかる．

図5は，図4の左上のT_1とT_2の部分をセマフォを利用した

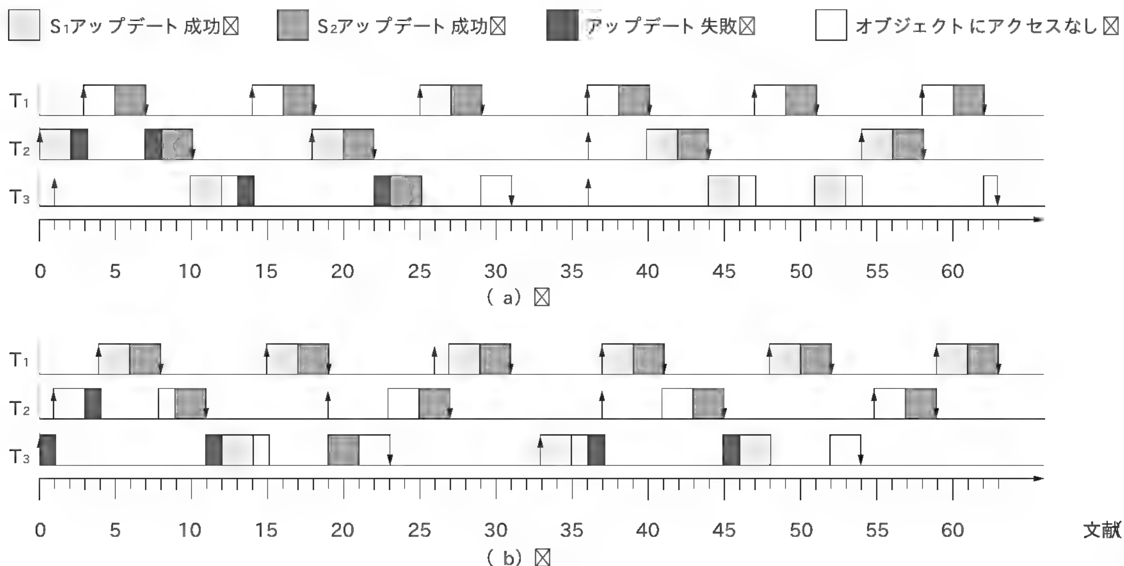


図4　ロック・フリー・プロトコルでのスケジューリングのようす

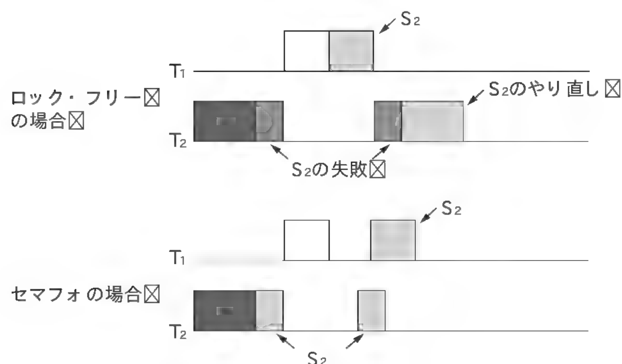


図5　セマフォを利用した場合との比較

場合と比較したものである．セマフォを利用した場合，T_1がS_2に入ろうとした時点でS_2はT_2によってロックされているためコンテキスト・スイッチが発生して，T_2に実行権が移る．この分，コンテキスト・スイッチが余計になる．図では，ロック・フリーの場合のほうがグレーのリトライしている部分があるので全体の時間が長くなっているように見えるが，実際はグレーの部分よりコンテキスト・スイッチにかかる時間のほうが長いことはすでに説明した．したがって，厳密に図を描けばロック・フリー・プロトコルのほうが早く終了することになる．

おわりに

余計なコンテキスト・スイッチが発生しないリトライ・ベースのリソース共有プロトコルついて紹介した．それぞれの関係をまとめると表1のようになる．

表1　各種プロトコルの比較

	プロトコル	リトライ回数	使用条件
セマフォ	ロック・ベースの排他制御	なし	なし
ブロック・フリー	リトライ・ベース	3	Writer-Reader
ロック・フリー	リトライ・ベース	1	最短連続実行時間保証

リトライ・ベースのデータ共有は，システム・コールなどを使用しない非常に単純な手法なので，メモリ共有型のマルチプロセッサ環境でも，単一CPUシステムでも，そしてRTOSなしでも使用できる．

参考文献

(1) H. Kopetz, J. Reisinger, *The Non-Blocking Write Protocol NBW: A Solution to a Real-Time Synchronization Problem*, Proceedings of the IEEE Real-Time Systems Symposium, pp. 131-137, 1993.

(2) J. H. Anderson, S. Ramamurthy, K. Jeffay, *Real-Time Computing Lock-Free Shared Objects*, Proc. Real-Time Systems Symposium, Pisa, Italy, pp.28-37, 1995.

ふじくら・としゆき　(株)豆蔵

バスマスタ転送用メモリ領域確保機能などを実装した

PCIデバッグ・ライブラリ for Win32 新バージョン登場

山際 伸一/丸山 治雄

1 PCI デバッグ・ライブラリとは

● ハードウェアが仕様通りに動作するか

PCI デバイスを設計開発して,「さぁ,Windowsでプログラム書くぞー」という段階になると,どうしてもデバッグのめんどうさに悩まされることが多々あるでしょう.また,自分が設計したデバイスではない場合,ハードウェアの動作を理解するために,レジスタの読み書き手順や割り込み制御レジスタの挙動など,一つ一つの動作を個別に確認したい場合もあるでしょう.

PCI デバイス内に実装したレジスタを正しく読み書きできるか,割り込みが発生したらステータス・ビットが立つかどうか,割り込み要求クリア処理を行ったらステータス・ビットがクリアされるか…といった,ハードウェアの基本的な動作を確認する必要があります.

PCI デバッグ・ライブラリは,このような状況を DLL とドライバで動的に吸収し,どのような PCI デバイスでも簡単にアクセスできるユーザ・インターフェースを提供するソフトウェア・パッケージです.

PCI デバッグ・ライブラリを用いたアプリケーションは,起動時に DLL をロードし,その DLL がドライバをダイナミックにカーネルに組み込み,Win32アプリケーションからデバイスにマップされた物理アドレス空間を容易にアクセスすることが可能になります.

このような方法は一般ユーザに対しては危険ではありますが,開発途中ハードウェアのデバッグという意味では非常に役に立つ方法です.したがって「PCI デバッグ・ライブラリ」はその名のとおり,デバッグをターゲットとしたソフトウェア・パッケージなのです.

このライブラリの実際の使い方などの詳しい解説は,参考文献 4)に掲載されているので,そちらを参照してください.

● 追加機能について

これまでの PCI デバッグ・ライブラリには次のような機能がありました.

(1) コンフィグレーション・レジスタ・アクセス機能(書き込み/読み出し)
(2) メモリ & I/O アクセス機能(シングル読み書き,ブロック読み書き,ブロック・コピー/フィル)
(3) 割り込みをユーザ空間へマップ

以上の機能で PCI デバイスのもつ基本的な機能の動作確認は可能ですが,いくつか動作確認のできない機能がありました.

PCI のメモリ・アクセスには,シングル転送とバースト転送があります.しかし,CPU が PCI メモリ空間にアクセスする場合は,基本的にはシングル転送しか発生しません.一般的にはポステッド・ライト・バッファ機能により,ライト動作の場合はバースト転送が発生しますが,リード動作の場合にはバースト転送が発生しません.

もう一つは仮想記憶に関する問題です.Windows は仮想記憶を採用しているので,Win32アプリケーション上でバッファ領域を確保しても,実際にはスワップ・アウトされて実メモリ上に存在しない場合があります.また,仮想メモリ空間は MMU によりページ単位で管理されているので,ページ・サイズを超えたサイズのバッファを確保しても,実メモリ上での物理アドレスは,ばらばらのアドレスに割り当てられる可能性もあります.これでは,バスマスタ対応デバイスに対して,マスタ・アクセス用のアドレスを指定することができません.

そのようなニーズに応え,次の二つの新機能を PCI デバッグ・ライブラリに追加します.

(1) SIMD 系命令によるバースト転送機能

マルチメディア処理用に最適化された MMX や SSE など SIMD 系命令を使うことにより,CPU が 64ビットや 128ビット・サイズでメモリ空間にアクセスすることができます.これらの SIMD 系命令では,一気にデータをメモリ・バスに書き出し,または読み込みをするので,ワード数は短いのですが,リード/ライトともにバースト転送を発生させることができます.

このアクセス先アドレスを PCI メモリ空間にすることで,PCI ターゲット・デバイスに対してバースト転送を発生させることが可能になります(ただし実際にバースト転送が発生するかどうかは,ホスト・ブリッジの仕様による).

表1　64/128ビット・アクセス関数

物理メモリ読み出し ULONG _MemReadLong(ULONG address) USHORT _MemReadShort(ULONG address) UCHAR _MemReadChar(ULONG address) ULONGLONG _MemReadLongLong(ULONG address)		
引き数	address	読み出す物理アドレス
戻り値	読み出しデータ	
機能	物理アドレスaddressの値を読み出す．_MemReadLongLongは64ビット幅，_MemReadLongは32ビット幅，_MemReadShortは16ビット幅，_MemReadCharは8ビット幅で読み出す	
UCHAR * _MemReadLongLongA(ULONG address, UCHAR *data) UCHAR * _MemRead128(ULONG address, UCHAR *data)		
引き数	address	読み出す物理アドレス
	data	読み出しデータを格納するバッファへのポインタ
戻り値	読み出しデータを格納したバッファへのポインタ	
物理メモリ書き込み void _MemWriteLong(ULONG address, ULONG data) void _MemWriteShort(ULONG address, USHORT data) void _MemWriteChar(ULONG address, UCHAR data) void _MemWriteLongLong(ULONG address, ULONGLONG data)		
引き数	address	書き込む物理アドレス
	data	書き込む値
戻り値	なし	
機能	物理アドレスaddressへdataの値を書き込む．_MemWriteLongLongは64ビット幅，_MemWriteLongは32ビット幅，_MemWriteShortは16ビット幅，_MemWriteCharは8ビット幅で書き込む	
void _MemWriteLongLongA(ULONG address, UCHAR *data) void _MemWrite128(ULONG address, UCHAR *data)		
引き数	address	書き込む物理アドレス
	data	書き込むデータを格納しているバッファへのポインタ
戻り値	なし	
物理アドレス・ブロック読み出し ULONG _MemReadBlockLong(ULONG address, UCHAR* data, ULONG count) ULONG _MemReadBlockShort(ULONG address, UCHAR* data, ULONG count) ULONG _MemReadBlockChar(ULONG address, UCHAR* data, ULONG count) ULONG _MemReadBlockLongLong(ULONG address, UCHAR* data,ULONG count) ULONG _MemReadBlock128(ULONG address, UCHAR* data, ULONG count)		
引き数	address	読み出す物理アドレス
	data	読み出しデータを格納するバッファへのポインタ
	count	読み出しワード数
戻り値	読み出しワード数	
機能	物理アドレスaddressからcountワードをdataの示すメモリに読み出す．_MemReadBlockLongでは32ビット幅で物理メモリにアクセスを行い，countのワード数は4バイトの転送を1ワードとする．以下，同様に，_MemReadBlockShortでは16ビット幅，2バイト転送で1ワード，_MemReadBlockCharでは8ビット幅，1バイト転送で1ワード．_MemReadBlockLongLongは64ビットで1ワード，_MemReadBlock128は128ビットで1ワード，戻り値のワード単位もそれぞれ4, 2, 1, 8, 16バイト	
物理アドレス・ブロック書き込み ULONG _MemWriteBlockLong(ULONG address, UCHAR* data, ULONG count) ULONG _MemWriteBlockShort(ULONG address, UCHAR* data, ULONG count) ULONG _MemWriteBlockChar(ULONG address, UCHAR* data, ULONG count) ULONG _MemWriteBlockLongLong(ULONG address, UCHAR* data, ULONG count) ULONG _MemWriteBlock128(ULONG address, UCHAR* data, ULONG count)		
引き数	address	書き込む物理アドレス
	data	書き込みデータを格納しているバッファへのポインタ
	count	書き込みワード数
戻り値	書き込みワード数	
機能	dataの示すメモリの値を，物理アドレスaddressへcountワード書き込む．_MemWriteBlockLongでは32ビット幅で物理メモリにアクセスを行い，countのワード数は4バイトの転送を1ワードとする．以下，同様にして_MemWriteBlockShortでは16ビット幅，2バイト転送で1ワード，_MemWriteBlockCharでは8ビット幅，1バイト転送で1ワード．_MemWriteBlockLongLongは64ビットで1ワード，_MemWriteBlock128は128ビットで1ワード．戻り値のワード単位もそれぞれ4，2，1，8，16バイト	
物理メモリ・コピー ULONG WINAPI _MemCopyLong(ULONG src_address, ULONG dest_address, ULONG count) ULONG WINAPI _MemCopyShort(ULONG src_address, ULONG dest_address, ULONG count) ULONG WINAPI _MemCopyChar(ULONG src_address, ULONG dest_address, ULONG count) ULONG _MemCopyLongLong(ULONG src_address, ULONG dest_address, ULONG count) ULONG _MemCopy128(ULONG src_address, ULONG dest_address, ULONG count)		
引き数	src_address	コピー元の物理アドレス
	dest_address	コピー先の物理アドレス
	count	コピーするワード数
戻り値	コピーしたワード数	
機能	物理アドレスsrc_addressからcountのワード数のメモリ内容をdest_addressの物理アドレスにコピーする．_MemCopyLongでは32ビット幅で物理メモリにアクセスを行い，Countのワード数は4バイトの転送を1ワードとする．以下，同様に，_MemCopyShortでは16ビット幅，2バイト転送で1ワード，_MemCopyCharでは8ビット幅，1バイト転送で1ワード．_MemCopyLongLongは64ビットで1ワード，_MemCopy128は128ビットで1ワード．戻り値のワード	

（2）マスタ・アクセス用メモリ確保機能

メイン・メモリ空間のスワップ・アウトしない領域にマスタ・アクセス可能なバッファ領域を確保し，その領域をユーザ空間からアクセス可能にする機能を追加します．

上記の二つの機能はあくまでも追加機能です．これまでのPCIデバッグ・ライブラリの機能は，過去のデバッグ用リソースを再利用できるように，新ライブラリに継承するようにします．

2 SIMD系命令によるバースト転送機能

● SIMD系命令

今回は64ビット・アクセスにはMMX命令を，128ビット・

表1　64/128ビット・アクセス関数（つづき）

物理メモリ・フィル ULONG WINAPI _MemFillLong(ULONG address, ULONG data, ULONG count) ULONG WINAPI _MemFillShort(ULONG address,USHORT data, ULONG count) ULONG WINAPI _MemFillChar(ULONG address, UCHAR data, ULONG count) ULONG _MemFillLongLong(ULONG address, ULONGLONG data, ULONG count) ULONG _MemFill128(ULONG address, ULONGLONG *data, ULONG count)		
引き数	address	フィルする先頭の物理アドレス
	data	書き込む値
戻り値	count	フィル・ワード数
	フィルしたワード数	
機能	物理アドレスからcountワードの領域に値dataを連続して書き込む．_MemFillLongでは32ビット幅で物理メモリにアクセスを行い，Countのワード数は4バイトの転送を1ワードとする． 以下同様に_MemFillShortでは16ビット幅，2バイト転送で1ワード，_MemFillCharでは8ビット幅，1バイト転送で1ワード．_MemFillLongLongは64ビットで1ワード，_MemFill128は128ビットで1ワード．戻り値のワード単位もそれぞれ4，2，1，8，16バイト	

ULONG	_CpuCheck()
引き数	なし
戻り値	ビット0‘1’のとき64ビット・アクセス可能 ビット1‘1’のとき128ビット・アクセス可能

リスト1　_CpuCheck()関数の使い方

```
ProcInfo = _CpuCheck();

if(ProcInfo & PCI_16BYTE_ACCESS_AVAILABLE){
  _MemReadBlock128(mem_base+i*BLOCK_SIZE, (UCHAR
                *)test_buf+i*BLOCK_SIZE, BLOCK_SIZE/16);
}
```

リスト2　CPUの種類判定

```
{
  ULONG ProcFeature;

  // CPUIDを調べます．
  // EAXに1をいれて，CPUID命令を実行すると，
            プロセッサの拡張機能の情報がEDXに返ります．
  __asm {
    mov eax, 0x1 ;
    cpuid ;
    mov ProcFeature, edx ;
  }
  if((ProcFeature & (1<<23)) != 0) //MMXが使えます．
    with_MMX = 1;
  if((ProcFeature & (1<<25)) != 0) //SSEが使えます．
    with_SSE = 1;
  if((ProcFeature & (1<<26)) != 0) //SSE2が使えます．
    with_SSE2 = 1;
}
```

アクセスにはSSE2命令を使っています．しかしMMXやSSE2命令をもたないCPUも存在するので，その有無をチェックする機能が必要になります（MMX命令はMMX Pentiumから，SSE2命令はPentium4から使える）．

MMX命令が使える場合，movq命令を用いることで64ビットのメモリ・アクセスが可能になります．一方，SSE2命令が使える場合，もっとも長いデータ転送を一括して行える命令であるmovd命令を用いることで，128ビットのメモリ・アクセスが可能です．64/128ビット・アクセス関数として，表1に示すライブラリ関数を追加しました．

それぞれの関数はWindows APIのDeviceIoControl関数でドライバを呼び出し，ドライバ内部でMMX/SSE2命令での転送を行うようにします．

なお，_MemReadLongLongA関数と_MemWriteLongLongA関数は，64ビット・データ型（LongLong型）が宣言できない言語を用いているときに使用します．たとえば，Visual Basic 6.0まではLongLong型がないようなので，バッファを用意してポインタを受け渡しして，データを読み書きします．

● _CpuCheck()関数

64/128ビット・アクセス関数を使う場合は，必ずCPUが16バイトまたは8バイト転送が可能かどうかを，_CpuCheck()関数で調べなくてはなりません．必死にPCIバスの波形を見て「バーストしないなぁ？」というときは，CPUが命令をサポートしていないことをはじめに疑いましょう．リスト1に示すように_CpuCheck()関数の戻り値により，64/128ビット・アクセス関数が使えるかどうかを判断してください．

● ドライバ内の実装の詳細

MMX/SSE2命令を用いることができるかどうかはドライバの中で判断させています．これは，非対応のCPUでこれらの命令を実行しないよう，変数with_MMXとwith_SSE2にフラグをセットするためです．

リスト2にCPUの種類を判定する例を示します．cpuid命令を用いて，MMX/SSE2命令の可否を調べています．

64ビット/128ビット転送はドライバの中で行います．リスト3に読み出しの際に用いられるドライバの中のPciIoctlReadMem関数の一部を示します．“case 8:”，“case 16:”は，関数の引き数として指定された転送単位でスイッチしているところです．ドライバの中でも，不具合が起きないように，_CpuCheck()関数が呼ばれた際にセットされた，“with_MMX”と“with_SSE2”を用いて，アクセスを制限しています．

ここで，16バイト・アクセスするためのコードで使う命令について解説します．SSE2ではmovdqa，movdqu命令のそれぞれが128ビットXMMレジスタを用いて，1命令で16バイトの転送を可能にしています．

しかし，movdqa命令の制限（p.149のコラム1参照）から，アドレスが16バイトでアラインされているか判断し，movdqu

リスト 3　64ビット/128ビット転送

```
    case 8: // 64ビット・アクセス

      //#ifdef __MMX__ // SSEではMOVDはXMMレジスタへの移動ができない
                                          ので，MMXレジスタを使います．
      if(with_MMX){
        if(STATUS_SUCCESS ==
                     KeSaveFloatingPointState(&float_save)){
          long     cnt;

          cnt = read_param->count;

          __asm{
            mov        ecx,cnt
            mov        esi,maped
            mov        edi,lpOutBuffer
            lea        esi,[esi + ecx * 8]
            lea        edi,[edi + ecx * 8]
            neg        ecx

            _MMX_loop:
                 movq  mm0,[esi + ecx * 8]
                 movq  [edi + ecx * 8],mm0
                 add        ecx,1
                 jnz        _MMX_loop

            emms
          }
          KeRestoreFloatingPointState(&float_save);
        }
        else{
          err = TRUE;
        }
      }
      else{
        err = TRUE;
      }
      //#endif
      break;
    case 16: //128 ビット・アクセス．MMX，SSEではできません．
      if(with_SSE2){
        if(STATUS_SUCCESS == KeSaveFloatingPointState
                                              (&float_save)){
          long     cnt;

          cnt = read_param->count;

          if(((ULONG)maped & 0xF)
                              && ((ULONG)lpOutBuffer & 0xF)){
           __asm{
             mov ecx,cnt
               shl ecx,4
               mov esi,maped
               mov edi,lpOutBuffer
               lea esi,[esi + ecx]
               lea edi,[edi + ecx]
               neg ecx

             _SSE2_loop_r0:
               movdqu   xmm0,[esi + ecx]
               movdqu   [edi + ecx],xmm0
               add      ecx,16
               jnz      _SSE2_loop_r0
               emms
            }
          }
          else if(!((ULONG)maped & 0xF)
                              && ((ULONG)lpOutBuffer  & 0xF)){
            __asm{
              mov ecx,cnt
                shl ecx,4
                mov esi,maped
                mov edi,lpOutBuffer
                lea esi,[esi + ecx]
                lea edi,[edi + ecx]
                neg ecx

                _SSE2_loop_r1:
                movdqa       xmm0,[esi + ecx]
                movdqu       [edi + ecx],xmm0
                add          ecx,16
                jnz          _SSE2_loop_r1

              emms
             }
          }
          else if(((ULONG)maped & 0xF)
                              && !((ULONG)lpOutBuffer & 0xF)){
            __asm{
              mov ecx,cnt
                shl ecx,4
                mov esi,maped
                mov edi,lpOutBuffer
                lea esi,[esi + ecx]
                lea edi,[edi + ecx]
                neg ecx

                _SSE2_loop_r2:
                movdqu       xmm0,[esi + ecx]
                movdqa       [edi + ecx],xmm0
                add          ecx,16
                jnz          _SSE2_loop_r2

              emms
             }
          }
          else {
            __asm{
              mov ecx,cnt
                shl ecx,4
                mov esi,maped
                mov edi,lpOutBuffer
                lea esi,[esi + ecx]
                lea edi,[edi + ecx]
                neg ecx

                _SSE2_loop_r3:
                movdqa       xmm0,[esi + ecx]
                movdqa       [edi + ecx],xmm0
                add          ecx,16
                jnz          _SSE2_loop_r3

              emms
             }
          }
        }
      }
```

命令とで用いる命令を選択しなければなければなりません．

3 マスタ・アクセス用メモリ確保機能

● マスタ・アクセスに必要な機能

マスタ・アクセスを許容するために必要な機能とは何でしょうか?

Windowsが動作している最中に，PCIデバイスが勝手にホスト・メモリをアクセスすると，当然ながら大惨事を招くだろうという予想がつく読者も多いでしょう．したがって，PCIデバイスが自由にアクセス可能な領域を作ってあげる必要があります．

また，PCIデバイスがメイン・メモリにアクセスする際には，PCIデバイスは物理アドレスでアクセスしなければなりません．しかし，Windowsのユーザ・アプリケーションとドライバは仮想アドレスで動作しており，直接PCIデバイスが仮想アドレスを用いることはできません．したがって，仮想アドレスから物理アドレスに変換する機能も必要になります．

さらにWindowsでは，不要なメモリ・ページをディスクに

COLUMN 1

128ビット転送するためのSSE2命令

SSE2命令で128ビット転送するには，

- movdqa，movdqu命令の組を使う
- movntdq命令を使う

の方法があります．movdqaとmovdquの違いは，16バイト・アラインされたメモリ・アドレスへの転送をするかどうかで使い分けます．movdqaがアラインされている場合の命令です．もし，movdqa命令にアラインされていないアドレスを渡すと，一般保護例外が発生してしまいます．しかしインテルの情報によると，「movdqu命令は実行速度が遅い」となっています．どのくらい遅いかは明文化されていません．

また，movntdq命令は基本的にはmovdqaと同じ動作をしますが，XMMレジスタ→メモリの転送のみです．この命令は，movdqa命令と一つだけ異なることがあります．Pentium4のメモリ・コンシステンシ・モデルはWeek Orderingなので，キャッシュからメモリへのアクセスが完了しているかどうかが明確にわかりません．通常，FENCE命令などを使って，メモリ・アクセスの完了を待たせる必要がありますが，movntdq命令はキャッシュをバイパスして，指定されたメモリに書き込みます．

しかし，これはマルチプロセッサ構成をとる場合で，さらに，キャッシュを有効にしている場合に影響があります．PCIデバッグ・ライブラリでは，キャッシュ禁止にしたPCIデバイスのメモリ空間にアクセスを発生させるので，このような問題は発生しません．

一時的に退避するページングを行っているために，PCIデバイスがアクセスしていた空間が次の瞬間にはディスクにスワップ・アウトされてしまう場合があります．これを回避するために，PCIデバイスのアクセスする領域だけは，スワップしないように，Windowsに通知する必要があります．

以上の三つの事柄を実現し，メモリ領域を確保できるような機能をPCIデバッグ・ライブラリに追加します．

● メモリ確保の動作

今回用いる方法は，図1に示すようにドライバがカーネル空間に連続領域を確保し，ユーザ空間からはDeviceIoControl関数を用いて，その領域を読み書きする方法です．この方法はスピード的には速い策ではないのですが，もっとも簡単に実現できます．

この方法を実現するためには，ドライバがロックされた連続メモリ領域を確保し，その先頭の物理アドレスを得られればよいわけです．

WindowsのカーネルコールであるMmAllocateContiguousMemory関数が連続領域の確保，ページングの禁止をした領域を提供します．この関数を用いて上記の条件に合う領域を作成します．この関数はページ・サイズ(4Kバイト)にアラインされたアドレスが返されるので非常に使いやすい領域を確保できます．さらに，確保するメモリのアドレスに制限を設けることもできます．2番目の引き数ではアドレス空間を制限するためのマスクを渡すことができます．これで，アドレス空間が制限される場合(たとえばISAデバイスなど)のデバッグも可能になります．

確保したメモリは連続領域であるため，先頭のアドレスだけがわかれば十分です．カーネル・コールのMmGetPhysicalAddress関数に領域の先頭アドレスを与えることで，物理アドレスを取得することができます．

領域の解放にはMmFreeContiguousMemory関数を用います．これは確保用関数と対になる解放関数です．

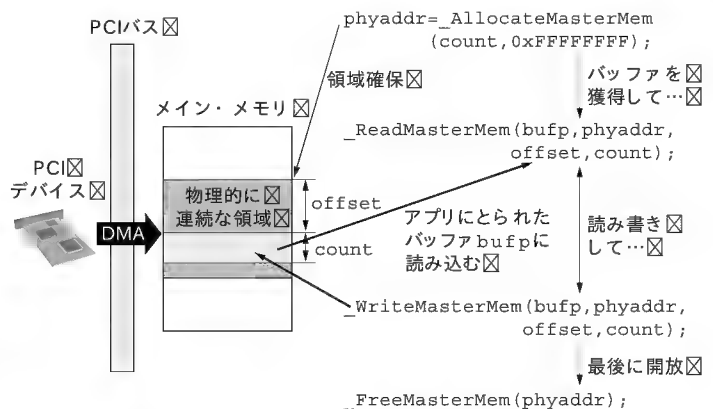


図1 メモリ確保の動作

● 使い方の例――メモリ確保

表2にマスタ・アクセス用のメモリ領域操作関数を示します．これらの関数は，Windows APIのDeviceIoControl関数を介して，ドライバにその操作を要求します．

これらの関数を用いる際には，C言語で用いられるmalloc，freeと同様の考えで用いることができるわけですが，領域への読み書きだけは，_ReadMasterMem関数，_WriteMasterMem関数を用いなければならないということを注意すれば，簡単に用いることができます．図1に使い方の例を示すので参考にしてください．

● ドライバ内の実装の詳細

表2に示した関数の機能はドライバ中で実現されています．DeviceIoControl関数でそれぞれの処理を呼び出しています．ドライバの中のPciDispatch関数でリスト4に示すように，メモリの確保，解放，読み書きを行っています．

マスタ・アクセス用に確保したメモリ領域をドライバは管理しています．これは，ユーザ・アプリケーションからの要求により確保したマスタ・アクセス用メモリ空間を，解放せずに終了する

リスト 4　メモリの確保，解放，読み書き

```
        case IOCTL_PCI_ALLOCATE_MASTER_MEM:
          Status = STATUS_UNSUCCESSFUL;
          if(pIrpStack->Parameters.DeviceIoControl.InputBufferLength >= sizeof(PCIDEBUG_MEMALLOC_INPUT)){
            PVOID buf_va = NULL;
            PHYSICAL_ADDRESS phyaddr, addrmask;
            PCIDEBUG_MEMALLOC_INPUT memlist;

            memlist.count = ((PCIDEBUG_MEMALLOC_INPUT *)pIrp->AssociatedIrp.SystemBuffer)->count;
            addrmask.LowPart = ((PCIDEBUG_MEMALLOC_INPUT *)pIrp->AssociatedIrp.SystemBuffer)->addrmask;
            addrmask.HighPart = 0;
            memlist.addrmask = addrmask.LowPart;
            if((memlist.count == 0) || (memlist.addrmask == 0)) break;

            if((buf_va = MmAllocateContiguousMemory(memlist.count, addrmask)) == NULL) break;
            memlist.vaddr = buf_va;
            phyaddr = MmGetPhysicalAddress(buf_va);

            ((PCIDEBUG_MEMALLOC_INPUT *)pIrp->AssociatedIrp.SystemBuffer)->phyaddr = phyaddr.LowPart;

            memlist.phyaddr = phyaddr.LowPart;

            // ここで、メモリ情報をリストに追加
            if(AddMasterMemList(memlist)!= STATUS_SUCCESS){
              MmFreeContiguousMemory(buf_va);
              break;
            }
            pIrp->IoStatus.Information = sizeof(PCIDEBUG_MEMALLOC_INPUT);
            Status = STATUS_SUCCESS;
          }
          break;
        case IOCTL_PCI_FREE_MASTER_MEM:
          Status = STATUS_UNSUCCESSFUL;
          if(pIrpStack->Parameters.DeviceIoControl.InputBufferLength >= sizeof(PCIDEBUG_MEMALLOC_INPUT)){
            PVOID buf_va;
            PHYSICAL_ADDRESS phyaddr, addrmask;
            PCIDEBUG_MEMALLOC_INPUT memlist;

            memlist.phyaddr = ((PCIDEBUG_MEMALLOC_INPUT *)pIrp->AssociatedIrp.SystemBuffer)->phyaddr;
            if(memlist.phyaddr == 0) break;

            DeleteMasterMemList(memlist);
            Status = STATUS_SUCCESS;
          }
          break;
        case IOCTL_PCI_READ_MASTER_MEM:
          Status = STATUS_UNSUCCESSFUL;
          if(pIrpStack->Parameters.DeviceIoControl.InputBufferLength >= sizeof(PCIDEBUG_MASTERMEMRW_INPUT) &&
          pIrpStack->Parameters.DeviceIoControl.OutputBufferLength >= sizeof(((PCIDEBUG_MASTERMEMRW_INPUT *)pIrp->
                                                                   AssociatedIrp.SystemBuffer)->count)){
            ULONG count, offset, phyaddr;
            PUCHAR outbuf, membase;
            ULONG copy_counter;

            count = ((PCIDEBUG_MASTERMEMRW_INPUT *)pIrp->AssociatedIrp.SystemBuffer)->count;
            offset = ((PCIDEBUG_MASTERMEMRW_INPUT *)pIrp->AssociatedIrp.SystemBuffer)->offset;
            phyaddr = ((PCIDEBUG_MASTERMEMRW_INPUT *)pIrp->AssociatedIrp.SystemBuffer)->phyaddr;
            outbuf = (PUCHAR)pIrp->AssociatedIrp.SystemBuffer;

            if((membase = FindMasterMem(phyaddr)) == NULL) break;
            for(copy_counter = 0; copy_counter < count; copy_counter ++)
              outbuf[copy_counter] = membase[copy_counter+offset];
            pIrp->IoStatus.Information = count;

            Status = STATUS_SUCCESS;
          }
          break;
        case IOCTL_PCI_WRITE_MASTER_MEM:
          Status = STATUS_UNSUCCESSFUL;
          if(pIrpStack->Parameters.DeviceIoControl.InputBufferLength == sizeof(PCIDEBUG_MASTERMEMRW_INPUT)){
            ULONG count, offset, phyaddr;
            PUCHAR inbuf, membase;
            ULONG copy_counter;

            count = ((PCIDEBUG_MASTERMEMRW_INPUT *)pIrp->AssociatedIrp.SystemBuffer)->count;
            offset = ((PCIDEBUG_MASTERMEMRW_INPUT *)pIrp->AssociatedIrp.SystemBuffer)->offset;
            phyaddr = ((PCIDEBUG_MASTERMEMRW_INPUT *)pIrp->AssociatedIrp.SystemBuffer)->phyaddr;
            inbuf = ((PCIDEBUG_MASTERMEMRW_INPUT *)pIrp->AssociatedIrp.SystemBuffer)->buf;

            if((membase = FindMasterMem(phyaddr)) == NULL) break;
            for(copy_counter = 0; copy_counter < count; copy_counter ++)
              membase[copy_counter+offset] = inbuf[copy_counter];
            pIrp->IoStatus.Information = count;

            Status = STATUS_SUCCESS;
          }
          break;
        }
```

COLUMN 2

Windows9x系OSの場合

丸山 治雄

● 開発に必要なツール

Windows9x用ドライバ(VxDタイプ)を開発するときは，コンパイラには次のように，少々古いものを使用します．最新の開発ツールでは作成できないので注意が必要です．ただしドライバを呼び出すライブラリやアプリケーションは，最新の開発ツールで問題ありません．

●DDK

Windows98用DDK(Win98用のDDKでリンクするとWindows95では動作できない)

●コンパイラ

MSVC2.0(デバイス・ドライバ用)

●リンカ

デバイス・ドライバをリンクするときにはMSVC2.0付属のリンカが必要

●アセンブラ

Windows98用DDKに付属のアセンブラを使用

今回のバージョン・アップではWinodws98用のDDKを使用したので，Windows95環境では動作できなくなりました．Windows95環境で動作させる場合は旧バージョンのPCIデバッグ・ライブラリを使ってください．

● MMX/SSE2命令について

WindowsNTでは，ドライバ内で64ビットおよび128ビット転送を実現していますが，Windows9xで動作させるきは，ライブラリ内で転送を行っています．これは，使用したコンパイラがこれらの命令群に対応していないためで，処理フローとしては見た目が悪いのですが，苦肉の策としてそうしています．

● メイン・メモリの確保

PCIDEBUG.DLLの_AllocateMasterMem()を使用してメイン・メモリを確保するときは，次の点に注意してください．Windows9xでは，システム上で動作しているドライバすべての合計が128Mバイトまでしかアロケートできません．テスト的に大きなメモリを確保してみるのはかまいませんが，正式な製品とするときは1Mバイト以下にするようにしてください．

また，WindowsNT用ドライバはWindows2000やWindowsXPでも動作しますが，WindowsXPは，基本的に，WDMでドライバを作成するように規定されています．WDMではこの関数は使用できないので注意してください．

なお，Windows98では特に制限がありませんが，ドライバがあまり大きな領域を占有すると，システムの動作に悪影響を与える恐れがあるので注意してください．

まるやま・はるお ドライバ屋

とメモリがリークしてしまうため，ドライバがアンロードされる際にすべて解放できるようにするためです．AddMasterMemList関数はメモリ情報をリンク・リストに追加します．DeleteMasterMemList関数はリンク・リストから一致するメモリ情報を削除し，さらにそのメモリ領域を解放します．

*　　　　*

以上の説明の中では，ドライバの内部的な動作はWindows2000やXPでの場合について説明しています．Windows98などで動作するドライバの作成については，Windows98用DDKなどが必要です(コラム2参照)．

ここで作成した新バージョンのPCIデバッグ・ライブラリ for Win32は，InterGiga No.34に収録し，さらに本誌Webページからダウンロードできるように準備する予定です．

参考文献

(1) IA-32 Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference，Intel，2003年．
(2) AP-485インテル プロセッサの識別とCPUID命令，Intel，2002年1月．
(3) Windows Driver Development Kit Document，Microsoft．
(4) PCIデバイス設計入門ーPCIバスの原理からHDLによるIC設計&デバッグ手法まで，TECH I(Vol.3)，CQ出版(株)．

やまぎわ・しんいち
まるやま・はるお ドライバ屋

表2 マスタ・アクセス用のメモリ領域操作関数

物理メモリ領域の確保 ULONG _AllocateMasterMem(ULONG count, ULONG addrmask)		
引き数	count	確保したいバッファのバイト数
	addrmask	確保したいアドレスのマスク指定
戻り値	確保したバッファの物理メモリ・アドレス(確保に失敗した場合は0)	
物理メモリ領域の解放 void _FreeMasterMem(ULONG phyaddr)		
引き数	phyaddr	_AllocateMasterMem関数で獲得した物理メモリ・アドレス
戻り値	なし	
確保した物理メモリ領域のブロック読み出し ULONG _ReadMasterMem(PUCHAR bufp, ULONG phyaddr, ULONG offset, ULONG count)		
引き数	bufp	読み出しデータを格納するバッファのポインタ
	phyaddr	物理メモリ・アドレス
	offset	オフセット
	count	読み出しバイト数
戻り値	読み出しバイト数	
確保した物理メモリ領域へのブロック書き込み ULONG _WriteMasterMem(PUCHAR bufp, ULONG phyaddr, ULONG offset, ULONG count)		
引き数	bufp	書き込みデータを格納したバッファのポインタ
	phyaddr	物理メモリ・アドレス
	offset	オフセット
	count	書き込みバイト数
戻り値	書き込みバイト数	

IPパケットの隙間から

ついにやってきた架空請求！

祐安 重夫

いろいろなところに届いているという噂は，前からよく聞いていた．最初のうちはこんなものが届いたなどという相談がインターネット上の掲示板に書き込まれると，けっこうみんなが真剣に答えてくれていたようだ．最近ではあまりにも数が多くなったせいか，せいぜい「そんなものは無視して放っておきなさい」という返事がちらほらあるか，ひどいときは質問自体が無視されてしまうほど，世の中に広まっているようだ．

そのわりにはうちにはやってこないし，一体どういうものなのか，実は興味津々でいた．もっとも，積極的に届いて欲しいものではないのはいうまでもないが，ある日郵便受けを開けてみたら，来ていた．「不良債権請求督促通達書」というはがきである．筆者は不謹慎なので，なんとなくわくわくしてしまった．いわずと知れた，架空請求という代物である．

それによると（原文のまま）「第１　債権に関する権利の変更現在貴方様の所有する不良債権の以降（ママ）に付きまして，運営業者様から債権譲渡に至りました」と最初から誤字が含まれているし，実際には印刷のレイアウトも変である．続いて「第２　裁判決議実施について期限までに解答がない債務者につきましては，裁判決議に同意したとみなし弊社顧問弁護士と共に協議の結果，最悪の場合裁判書の許可の下に担当回収員がご自宅に直接お伺いします．ご自宅不在の場合はお客さまの身辺調査を行い，会社の給料やご本人様の財産などの差し押さえ手続きを行わせていただきます．和解交渉の意思のある方は，大至急ご連絡していただけるようおねがいします」とあり，文法的に怪しかったり論理的にもおかしかったりする．

御丁寧にそれに続けて，赤字で「この通達書は最終通告になります」と印刷してある．2色刷りとは金をかけているなどとは，今どきだれも思わない．はがきは「インクジェット紙はがき（再生紙）」と書かれた官製はがきである．

もちろん「担当者直通」の電話番号は，5件印刷されていたが，すべて080の携帯電話である．こういう場合に業者をからかうために，わざわざプリペイド携帯を用意して電話をかけてみて，徹底的に相手をからかって遊ぶという暇人もいるという噂だが，筆者はそんな暇人ではないし，それ以前にプリペイド携帯を用意するようなそんなむだな金も持っていない．

しかし一応相手の会社名「（株）神×債権管理センター」が明記されていたので，Googleで検索してみたところ，各地の消費者センターなどがヒットした．

テレビのニュースなどがこういう問題を取り上げるのは，筆者のところにまではがきが届くくらい事件が広まってからと，なかなか時間がかかる．しかし，先日テレビを見ていたら，この件について取材していた．それによると，業者の住所として印刷された場所を訪ねてみると，そんな住所は存在しない，そこにはまったく関係のない別の会社が存在したといったもののほかに，会社がビルの60階以上にあるはずなのに，そこには9階建のビルしか存在しなかった例もあったそうだ．

筆者のところに届いたはがきの相手の住所は，存在しない住所という，もっとも単純なパターンだった．

当然だが，このはがきは無視したままだが，だれも訪ねてこないし（もっともうちの玄関には，普通のセールスマンでもそのまま引き返すような貼り紙が貼ってあるが），それ以上は何も起こっていない．

ところで，こういう時代なので，架空請求の類ははがきだけではなく，電子メールでも横行しているようだ．とくに携帯メールでの（アダルト・サイトなどからの）請求が，話題になることが多い．筆者の場合，携帯電話は一応持っているが，持っていることを公言するのさえこの文章が初めてだし，持っていることを知っているのは数人，電話番号やメール・アドレスを知っているのは一人だけという状態なので，こちらの被害にあったことは一度もない．携帯電話の料金も，毎月の基本料金以上は支払ったことがない．

それでは普通の電子メールはどうだろうかといえば，こちらは各国からのSPAMやウィルスの山なのだが，不思議と架空請求は来たことがない．中国語のメールは読めないので，最初からフィルタではじいているが，英語だけでもバイアグラから学位まで各種の怪しい通販をはじめとして，わけのわからないメールが山のように来るが，日本語のSPAMは比較的少ないようだ．

国民生活センターのサイト（http://www.kokusen.go.jp/）を見ると，電子メールによる架空請求というのも見境がなくて，国民生活センターのwebmasterにまで請求メールが届いているという．収集したメール・アドレスに対して，プログラムで自動で送信しているのだろうが，go.jpには送らないとかwebmasterのようなアドレスはスキップするという程度のくふうもないらしい．

しかし，この手のメールが届かないということは，筆者の会社や筆者個人のWebページは，SPAMのアドレス収集の対象になっても，架空請求の対象とはみなされていないということなのだろうか．これでも10年以上の歴史のあるドメインなのだが，お金があるとは思われていないのだろう（実際，貧乏だが）．

すけやす・しげお　インターメディア・アクセス

割り込みプライオリティを最適化して

SH-Linuxの割り込みレイテンシを改善

海老原 祐太郎

組み込みシステムのOSとしてLinuxを採用するにあたり，Linuxのリアルタイム性能が問題になる場面があります．たとえば，ハードウェアを直接制御しているプログラムでは，割り込み処理の遅延や取りこぼしが原因で致命的な不具合を生じる危険性があります．このような場面では，RTLinuxや先月号で取り上げたOCERAなど，Linuxのリアルタイム化ソフトウェアを採用するという方法があります．しかしPCアーキテクチャの事情と異なり，組み込みシステムのアーキテクチャでは，原則としてこれらリアルタイム化ソフトウェアの移植を自分で行う必要があります注1．

現状では，OCERAはSH-Linuxに移植されていません．しかし，せっかくSHプロセッサという組み込みシステムの制御に適したプロセッサを用いるのならば，Linuxから直接ハードウェアをコントロールし，I/O制御やネットワーク，ファイル・システムといった上位層までを一つのOSの下で構築したいところでしょう．

本稿では，筆者が比較的簡単なアイデアで，SH-Linuxの割り込みレイテンシの改善を行ったので，それを紹介します．

割り込みレイテンシの原因と解決策

割り込みレイテンシとは，ハードウェア割り込みが発生してから実際に割り込みルーチンに処理が移行するまでの遅延時間のことです．割り込みレイテンシが短いほうが良いことはいうまでもありません．割り込みレイテンシが長い状態では，ハードウェアからの要求に迅速に対応することができず，入力から出力までのフィードバック・ループが長くなったり，最悪の状態では割り込みを取りこぼすことも考えられます．

割り込みレイテンシが長くなる原因は，Linuxの既存のデバイス・ドライバが割り込み禁止にしている時間が比較的長いということが考えられます．たとえば，IDEのデバイス・ドライバが一時的に割り込み禁止状態で走行するために，ファイル入出力時にI/O制御の割り込みレイテンシが増加してしまうといったことが挙げられます．デバイス・ドライバが割り込み禁止にする理由は，CPUの性能をフルに使って処理時間を短くすることが目的ではなく，ほかのタスクとのレース・コンディション(競合状態)を回避することが目的です．わかりやすい例を示します．

```
1: tmp = inb(データ・レジスタ・アドレス);
2: tmp |= 0x80;
3: outb(tmp,データ・レジスタ・アドレス);
```

これはデータ・レジスタのビット7を1にする例です．C言語で記述すれば3行だけですが，この3行の間に割り込みが発生し，割り込みプログラム側で同じデータ・レジスタの書き換えが発生すると，3行目でデータ・レジスタの上書きが発生してしまうことになり，意図したとおりの動きをせずにバグの原因となってしまいます．このような場面では，割り込みを禁止し，3行が必ず「連続して」操作されるようにプログラムを記述します．プログラマが割り込みを禁止する意図は，文字通り「割り込まれては困るから」割り込みを禁止するのです．いいかえれば，まったく関係のないほかのI/O操作ならば割り込まれてもかまわないが，同じI/Oを操作するほかのタスクに割り込まれては困るのです．

RTLinuxやOCERAの基本的なアイデアは，OSをハイブリッド構造にし，ハードウェア的には割り込み禁止状態をつくることはせず，つねに割り込み可能な状態とするものです．ハードウェアからの割り込み要求は，いったんリアルタイムOS側で受け取る構造になっています．そしてLinuxの割り込み禁止から割り込み解除までをRTOS側でフックし，割り込み禁止(割り込まれては困る状態)の間は割り込みを保留します．そして，Linux側が割り込み可能な状態になった時点で，保留しておいた割り込みをLinuxに通知します．このアイデアは，特殊なハードウェアを用いなくてもすべてソフトウェアで完結している点が優れていると考えられます．

もちろん，RTLinuxやOCERAをSH-Linuxに移植することも可能ですが，SHプロセッサは比較的I/O制御向きに設計されていることもあり，優先度マスク付き割り込みコントローラが内蔵されているので，この割り込みコントローラの機能を生かす方向を考えてみます．

注1：オープン・ソースなのでプロジェクトを組んで開発を行うのがベストだが，だれかが始めなければならない．

表1　実験で使用したマイコン・ボード CAT709の仕様

CPU	SH7709S(SH-3) 117MHz
メイン・メモリ	SDRAM 32Mバイト
フラッシュ・メモリ	8Mバイト
バックアップ・メモリ	SRAM 512Kバイト
インターフェース	100Base Ethernet × 1
	シリアル× 3
	DIO
そのほか	CFソケット 時計用IC

表2　割り込みフラグの一覧

IRQ番号	割り込みタイプ	名　前	フラグ
9	IPR-IRQ	ide2	SA_INTERRUPT | SA_RESTORER
16	IPR-IRQ	timer	SA_INTERRUPT
23	IPR-IRQ	sci	SA_INTERRUPT
24	IPR-IRQ	sci	SA_INTERRUPT
25	IPR-IRQ	sci	SA_INTERRUPT
35	IPR-IRQ	NE2000	なし
36	IPR-IRQ	rtc	SA_INTERRUPT

表3　**SA_INTERRUPT**を付けた場合と付けなかった場合

request_irp()時にSA_INTERRUPTを付けた場合
●多重割り込みを許可しない ●割り込みサービス・ルーチンが「割り込み禁止状態」で実行される
SA_INTERRUPTを付けなかった場合
●多重割り込みが許可される ●割り込みサービス・ルーチンが「割り込み許可状態」で実行される．ただし，ドライバ自身の割り込みはマスク状態におかれるので，割り込みサービス・ルーチン自身がリエントラントである必要はない

Linuxでは幅広いアーキテクチャへの移植を考慮して，このようなプロセッサ固有の機能は使わないように設計されています．そしてSH-LinuxでもSHプロセッサの優先度付き割り込みコントローラは積極的には使用していません．しかし，せっかくプロセッサに内蔵されている機能なので，ここでは積極的に活用してみることにしました．

今回の実験で用いた組み込みLinux対応マイコン・ボードCAT709の仕様を表1に，外観を写真1に示します．なお，SHプロセッサ固有のCPU内蔵機能を活用するため，ほかのアーキテクチャ(PCなど)への移植は考慮していません．

既存の問題点

● 割り込み禁止時間

Linuxは割り込みレイテンシが比較的長いといわれています．その原因として，前述したようにIDEなどの既存のデバイス・ドライバが割り込み禁止にして走行している時間が比較的長いことが挙げられます．実際に今回実験に使用したLinuxマイコン・ボードで割り込みレイテンシを測定すると，最悪値として



写真1　実験で使用したマイコン・ボード CAT709[シリコンリナックス(株)]の外観

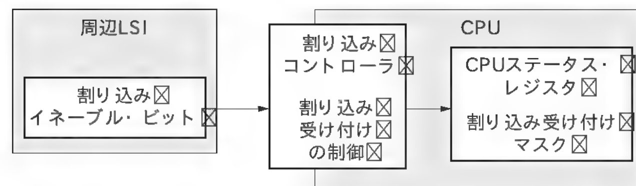


図1　一般的な割り込みマスク

2.5msを観測しました．これは，すなわち2.5ms以下の周期で発生している割り込みは，取りこぼす可能性があることを示しています．まずは割り込み禁止期間中に発生した割り込みの処理を改善します．

● 多重割り込み

第二の問題として，既存のデバイス・ドライバが割り込みルーチンを登録する際に「多重割り込みを許可しない」フラグで割り込みルーチンを登録していることが挙げられます．多重割り込みを行っても問題が発生しないだろうと考えられる場面でも，安全を見越して「多重割り込みを許可しない」フラグを立てて割り込みルーチンを登録しているデバイス・ドライバが多くあります(表2)．この「多重割り込みを許可しない」フラグはSA_INTERRUPTと定義されています(表3)．実際には割り込みルーチンそのものの処理は短時間と考えられるので，多重割り込みモードに変更したところで効果は現れないかもしれませんが，ここの処理も変更することにします．

割り込みマスク

割り込み禁止(cli()関数)，割り込み許可(sti()関数)はどのように割り込みをマスクしているのでしょうか．アーキテクチャにより実装は異なりますが，一般的にはプロセッサ内蔵(もしくは周辺)の割り込みコントローラの機能により割り込み

マスクを実現しています(図1).

一般的な割り込みアーキテクチャとSHプロセッサ固有の事情をあわせて記述します．周辺装置からは，割り込みが発生します．割り込みを発生させる周辺LSIのレジスタの中には，割り込みイネーブル/ディセーブルのビットが必ず存在します．大部分のLSIは，リセット解除状態の初期値では割り込みがディセーブル状態で，コントロール・ソフトウェアがレジスタを初期化する最終段階で割り込みをイネーブル状態にします．このように，周辺LSIのレジスタの中にも割り込みをコントロールするビットがありますが，その存在と制御方法についてはデバイス・ドライバしか知るよしがありません．OSからデバイス・ドライバに対して割り込みの有効/無効をコントロールする共通インターフェースが定義されていれば良いのですが，Linuxにはそのようなデバイス・ドライバに共通のインターフェースはありません．

周辺からの割り込み要求をプロセッサにどのように伝えるかを制御する回路が割り込みコントローラです．簡単な回路においては，ANDゲートだけで構成されています．周辺回路からの割り込み要求をCPUに伝えるのか，それともマスクするのかをコントロールします．エッジ出力の割り込みの場合は，割り込みコントローラがラッチを行います．

SHプロセッサにはCPU内部に割り込みプライオリティ・コントローラ回路が内蔵されています(図2)．周辺回路(この場合，シリアルなどの内蔵周辺回路も周辺回路とする)からの割り込み線を受け付け，1～15のプライオリティを付加してCPUに通知する機能をもっています．プライオリティ15が優先順位が一番高く，1が一番低くなっています．0に設定すると割り込みを通知しない，すなわち割り込みマスク状態となります．

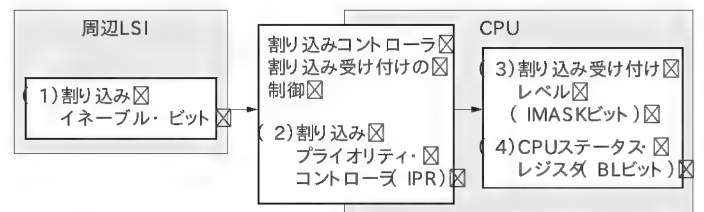
ほとんどのCPUには，CPU内蔵のステータス・レジスタ(CPUによって名称は異なる)に，割り込みを受け付けるか否かを示すフラグ・ビットがあります．SHプロセッサではステータス・レジスタ(SRレジスタ)にBLビットとして存在します(図3)．SHプロセッサは，割り込みを受け付けるとハードウェア的にBLビット=1として割り込みをブロックする状態になりますが，BL=1の状態では例外もマスクされてしまうため，SH-Linuxではこのビットはあまり使用していません．

ここからはSHプロセッサ固有の機能になりますが，SHプロセッサではステータス・レジスタ(SR)の中にIMASKと呼ばれる割り込みレベルを設定するビットが存在します(図3)．IMASKを設定することにより受け付け可能な割り込みレベルを調整することができます．IMASK=0の状態では，すべてのレベル(レベル1～15)の割り込みを受け付けます．IMASK=1の状態では，レベル1の割り込みをマスクし，レベル2～15の割り込みを受け付けます．IMASK=15の状態では，すべての割り込みをマスクします．割り込み優先度という考え方は，I/O制御の際のプロセッサに固有な概念なので，すべてのアーキテクチャに存在するとは限りません．Linuxカーネル自体にも割り込み優先度といった概念はありません．このため，ノーマルなSH-LinuxではIMASKを0もしくは15の2値で割り込みを制御し，割り込み禁止(`cli()`関数)，割り込み許可(`sti()`関数)としています．

注2：おそらくSHプロセッサにSH-1やSH-2と同様な割り込みベクタ・ジャンプ機能があったとしても使用しないだろう．

Linuxの割り込みの流れ

それでは実際にLinuxでの割り込み処理の流れを図4に従って解説します．Linuxでの割り込み処理のフローはアーキテクチャに依存せず，PCもSH-Linuxも同様です．いいかえれば，移植性や互換性を重視し，アーキテクチャ固有の最適化は行っていないのです．SHプロセッサでは，割り込みや例外要因によってハードウェア的なエントリ・ポイントは異なっています．しかし，このような特定のアーキテクチャ固有の機能は用いないため，すべての割り込みは同じアドレスに集められます[注2]．



(1)周辺LSI内部の割り込みイネーブル・ビット
　デバイス・ドライバしかその操作方法を知らない
(2)割り込みプライオリティ・コントローラ
　入力されてくる割り込みの優先度を決定する．優先度0にすると割り込みを受け付けない
(3)割り込み受け付けレベル
　CPUステータス・レジスタ内にある現在の割り込み受け付けレベル状態(0～15)
(4)割り込み・例外受け付け
　CPUステータス・レジスタ内にある割り込み・例外の受け付け/マスクを示すビット・ブロック状態にすると例外も受け付けなくなる

図2　SH-Linuxでの割り込みマスク

31	30	29	28	27 ～ 16	15	14	13 ～ 10	9	8	7 ～ 4	3	2	1	0
0	MD	RB	BL	0 ------------ 0	CL	0	0	M	Q	IMASK	0	0	S	T

MD　：モード・ビット
　　1=特権モード
　　0=ユーザ・モード
RB　：レジスタ・バンク切り替え
BL　：ブロック・ビット
　　1=例外・割り込みをブロック
　　0=ブロックしない
CL　：キャッシュ・ロック・ビット
　　1=キャッシュ・ロック使用
M，Q：算術演算で使用
IMASK：割り込みマスク・ビット
　　0：レベル1～15(全レベル)割り込み受け付け
　　1：レベル2～15割り込み受け付け
　　2：レベル3～15割り込み受け付け
　　15：全レベル割り込みマスク
S，T　：算術演算で使用

図3　SH-3ステータス・レジスタ

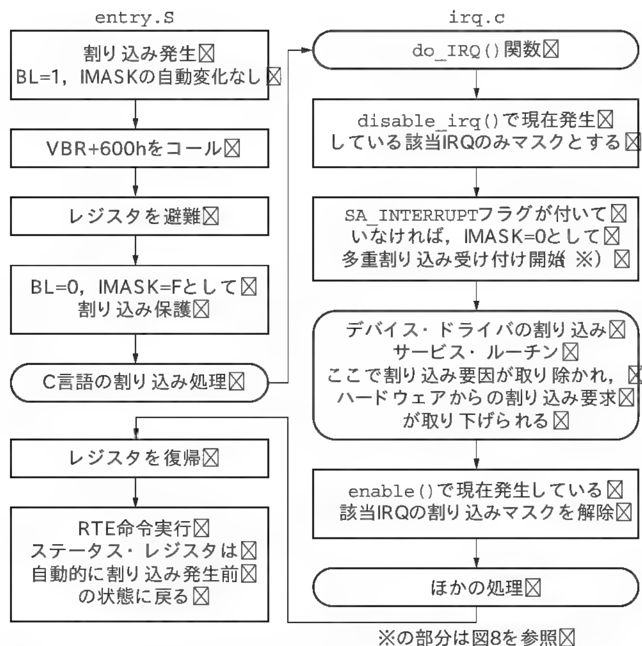


図4　Linuxの割り込み処理の流れ



図5　割り込み禁止時間

割り込みが発生するとアセンブラで記述されたエントリ・ポイントから処理が開始され，スタックの退避を行った後，C言語のdo_IRQ()関数が呼び出されます．

do_IRQ()に制御が移ると，BL=0，IMASK=Fとして割り込みマスクを設定しなおします．前述したようにBL=1では割り込みだけではなくMMU例外などもマスクされてしまうために使用しにくいのです．

その後，割り込みコントローラの機能を使用して現在発生している割り込みの優先度を0として，マスク状態にします．この状態でSA_INTERRUPTが設定されていなければ，多重割り込み許可ということで，sti()を実行します．すなわちIMASK=0とします．

周辺装置からは割り込み信号が発生している状態ですが，該当割り込みはマスク状態にしているので，sti()を実行しても同一割り込みの無限割り込みは発生せず，安全です．

ここまでをSH-Linuxカーネルが実行した後，デバイス・ドライバの割り込みサービス・ルーチンが呼び出されます．デバイス・ドライバの割り込みサービス・ルーチンでは，割り込み要因を調べ，該当割り込みをリセットします．これでようやく周辺装置からの割り込み要求が取り下げられます．

割り込みマスクを改善

前述したように，SHプロセッサには周辺装置からの割り込みを優先度1～15のレベルに設定することができます．しかし，このようなプロセッサ固有のしくみはLinuxでは採用していません．標準のSH-LinuxカーネルではIMASK=0もしくは15の2値で割り込みをマスクしています．実際にSH-Linuxカーネル・ソースを調べてみると，割り込みコントローラに設定する周辺装置からの割り込み優先度レベルは1か2，最大でも7といった値が採用されています．そこで，SH-Linuxで用いる割り込み優先度は1～8の範囲ということにします．cli()関数（割り込み禁止）ではIMASK=8とし，優先度1～8の割り込みをマスクします．優先度9～15の割り込みは割り込み禁止状態でもマスクしません．sti()関数（割り込み許可）ではIMASK=0とし，すべての割り込みを許可します（図5）．こうすることにより，割り込みレベル9～15は，Linuxの状態によらず，つねに割り込みの受け付けが可能な状態になります．割り込みレベル9～15は「高優先割り込み」と定義します．注意点としては，前述したようにLinuxは「割り込まれては困るから」割り込み禁止状態にするわけなので，「高優先割り込み」からはLinuxの関数を呼び出してはいけません．

図6に示すように，cli()～sti()はネストすると不具合を生じるので，Linuxではsave_and_cli(flags)～restore_flags(flags)を用います（図7）．このときも同様にsave_and_cli(flags)でIMASK=8とし，現在の割り込みレベルをflagsに保存します．restore_flags(flag)では保存されていた割り込みレベルを回復するようにします．

優先度継承多重割り込み

前述のように，SA_INTERRUPTフラグが付いていないときに発生している割り込みは，マスクした後にsti()によって割り込み許可の状態になります．この状態ではIMASK=0になるので，割り込みレベルとは無関係に，すべての割り込みを許可した状態になります．割り込みレイテンシの改善とは直接は関係ありませんが，無差別に割り込み可能な状態にするのではなく，現在発生している割り込み優先度を継承する形で割り込みレベルを設定します．すなわち割り込みレベル2が発生している状態では，割り込みレベル3～15が多重割り込み可能ということにします．

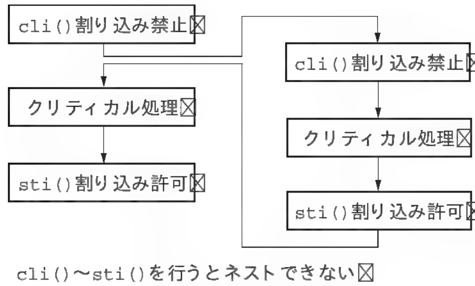


図6
割り込み禁止
処理1

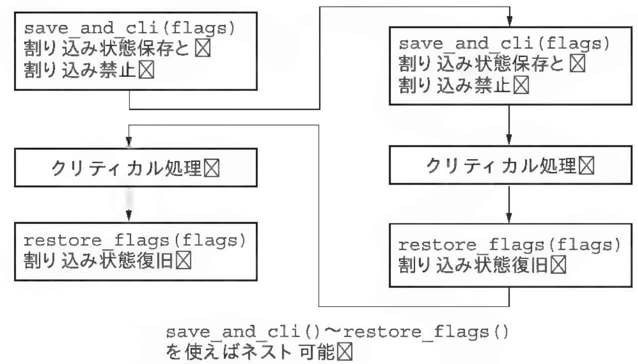


図7　割り込み禁止処理2

SA_INTERRUPTが付加されていない状態では，多重割り込みを許可していません．このときはIMASK=8として，Linuxのドライバで使われている優先度1〜8の割り込みをマスクして多重割り込みを許可しません．ただし，「高優先割り込み」は多重割り込み可能とします(図8).

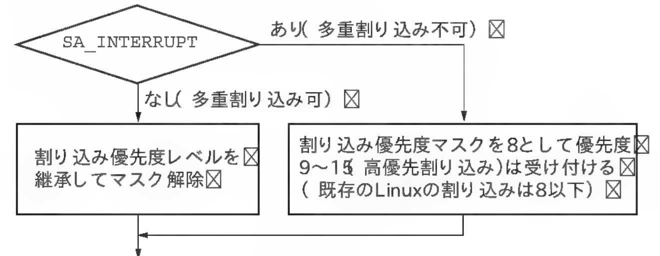


図8　割り込みマスクの優先度継承と高優先割り込み

カーネル・ソース・コードの変更

筆者はLinux-2.4.21カーネルを元に改善を行いました．arch/sh/kernelディレクトリ以下のirq.c，それからinclude/asm-sh/system.hに対して変更を加えました．Linuxカーネルのライセンスは GPLなので，変更後のカーネル・ソースもGPLとして筆者のWebサイト(http://www.si-linux.com/)からダウンロード可能です．

導入を行う場合には，筆者のWebサイトからカーネル・ソースをダウンロードしてください．通常のカーネル・ビルドと同じ手順で使えるようになります．

```
make config
make dep
make zImage
make module
```

リスト1　変更を加えたsystem.hの一部

```
/* Interrupt Control */

#define MAX_IRQ_LEVEL 8

#define set_ireg(ip)                                          ¥
do{                                                           ¥
  unsigned long __dummy;                                      ¥
  __asm__ __volatile__("stc      sr, %0¥n¥t"                  ¥
                       "and      %1, %0¥n¥t"                  ¥
                       "or       %2, %0¥n¥t"                  ¥
                       "ldc      %0, sr¥n"                    ¥
                       : "=&z" (__dummy)                      ¥
                       : "r" (~0xf0), "r" ((ip) & 0xf0)       ¥
                       : "t");                                ¥
}while(0)


#define __save_flags(x) ¥
        __asm__ __volatile__("stc sr, %0¥n¥t" ¥
                             "and #0xf0, %0"  ¥
                             : "=&z" (x) :/**/: "memory" )

static __inline__ void __cli(void)
{
  set_ireg(MAX_IRQ_LEVEL << 4);
}

static __inline__ void __real_cli(void)
{
  set_ireg(15 << 4);
}

static __inline__ void __sti(void)
{
  set_ireg(0);
}

static __inline__ unsigned long __save_and_cli(void)
{
  unsigned long flags;
   __save_flags(flags);
   __cli();
   return flags;
}

static __inline__ unsigned long __save_and_real_cli(void)
{
  unsigned long flags;
   __save_flags(flags);
   __real_cli();
   return flags;
}

#define __save_and_sti(x)       do { __save_flags(x); __sti(); }
                                                         while(0);

#define __restore_flags(x) set_ireg(x)

static  __inline__ void set_interrupt_priority(int priority)
{
  set_ireg(priority<<4);
```

リスト 2　割り込み発生から割り込みルーチンの入り口までの遅延時間を測定するプログラム **warikomi_latency.c**

```
/*
 * CAT709 Interrupt latency "warikomi_latency.c"
 *   for linux-2.4.x siries kernel
 *   Copyright Linux-koubou (si-linux.com)
 *   written by Y.Ebihara
 *         Nov/2001
 *         Aug/2004
 *
 */
#define MODULE
#define __KERNEL__

#define DEVNAME             "timer1"
#define TIMER1_IRQ 17

#include <linux/module.h>
#include <linux/fs.h>
#include <linux/sched.h>
#include <linux/proc_fs.h>
#include <asm/system.h>
#include <asm/uaccess.h>
#include <asm/io.h>

/* ---- timer controler --- */

#define TOCR  0xfffffe90  /* byte  */
#define TSTR  0xfffffe92  /* byte  */

/* ch 0 */
#define TCOR0 0xfffffe94  /* long  */
#define TCNT0 0xfffffe98  /* long  */
#define TCR0  0xfffffe9c  /* short */
/* ch 1 */
#define TCOR1 0xfffffea0  /* long  */
#define TCNT1 0xfffffea4  /* long  */
#define TCR1  0xfffffea8  /* short */
/* ch 2 */
#define TCOR2 0xfffffeac  /* long  */
#define TCNT2 0xfffffeb0  /* long  */
#define TCR2  0xfffffeb4  /* short */
#define TCPR2 0xfffffeb8  /* long read-only */


static unsigned long module_clock;      /* モジュール・クロック(Hz)*/
static unsigned long timer_clock;
                                  /* TIMER1ユニットへの入力クロック(Hz)*/
static unsigned long warikomi_interval=0;
                                          /* 割り込みインターバル設定値 */
static unsigned long delay_tick;        /* 前回の割り込みレイテンシ */
static unsigned long max_delay_tick=0;
                                          /* 割り込みレイテンシの最大値 */
static unsigned long overrun_count=0;
                                              /* 割り込みオーバラン回数 */

/* --------- ^-^ -------------------------------------------- */
/*  timer1 割り込みルーチン                                  */
/* --------- ^-^ -------------------------------------------- */

static void tmu1_handler(int irq, void *dev, struct pt_regs
                                                          *regs)
{
  /* 割り込みがかかってから、どのくらいタイマが進んだか */
  delay_tick=0xffffffff-ctrl_inl(TCNT1);
  if(max_delay_tick < delay_tick){
    max_delay_tick = delay_tick;
  }
  if(delay_tick > warikomi_interval){
    /* オーバランした */
    overrun_count += delay_tick/warikomi_interval;
  }
  /* タイマ1 割り込みフラグ・リセット */
  ctrl_outw(0x0021, TCR1);                      /* was 0x0021 */
  ctrl_outl(0xffffffff, TCOR1);
                              /* リロード・レジスタにインターバル・セット */
  ctrl_outl(warikomi_interval, TCNT1);
                         /* カウントダウン・レジスタにインターバル・セット */
}

/* --------- ^-^ -------------------------------------------- */
/*  timer1 初期化ルーチン                                    */
/* --------- ^-^ -------------------------------------------- */

static void init_timer1(unsigned long timer1_hz){
  module_clock = 29421200;
  timer_clock = module_clock/16;
  warikomi_interval = timer_clock / timer1_hz;
                                  /* Timer1に設定するインターバル値*/

  /* ポート D0ビットを出力とする */
#define PDCR 0xa4000106
  ctrl_outw(0xaa89,PDCR);
                           // (初期値は 0xaa8a)  0b 1010 1010 1000 1001

  printk("<1>module_clock=%d¥n",(int)module_clock);
  printk("<1>timer_clock=%d timer1_hz=%d
              interval=%d¥n",(int)timer_clock,(int)timer1_hz,(i
                                        nt)warikomi_interval);

  /* タイマ 1 停止 */
  ctrl_outb(ctrl_inb(TSTR)&~0x2,TSTR);   /* ch1 カウントダウン停止 */

  /* タイマ1 スタート */
  ctrl_outw(0x0021, TCR1);        /* 割り込み有効、プリスケーラ =1/16 */
  ctrl_outl(0xffffffff, TCOR1);
                              /* リロード・レジスタにインターバル・セット */
  ctrl_outl(warikomi_interval, TCNT1);
                         /* カウントダウン・レジスタにインターバル・セット */
  ctrl_outb(ctrl_inb(TSTR)|0x2, TSTR);  /* カウントダウン・スタート */
  request_irq(TIMER1_IRQ,tmu1_handler,0,DEVNAME,0);
                                  /* カーネルに割り込みルーチン登録 */
  printk("tmu1_handler=%p¥n",tmu1_handler);
}

int timer1_readproc(char *buf, char **start, off_t offset,
                              int count, int *eof, void *data
){
  char *temp = buf;
  unsigned long long delay_time;
  unsigned long long max_delay_time;

  disable_irq(TIMER1_IRQ);
  delay_time = delay_tick;
  max_delay_time = max_delay_tick;
  enable_irq(TIMER1_IRQ);

  delay_time *= 10000000;
  delay_time /= timer_clock;
  max_delay_time *= 10000000;
  max_delay_time /= timer_clock;

  temp += sprintf(temp,
                 "delay=%d tick (%12d.%1d usec) ,
                  max=%d tick (%12d.%1d usec) , overrun=%
d¥n",
                 (int)delay_tick,
                 (int)(delay_time/10),
                 (int)(delay_time%10),
                 (int)max_delay_tick,
                 (int)(max_delay_time/10),
                 (int)(max_delay_time%10),
                 (int)overrun_count);
  *eof=1; /* EOF */
  return temp-buf;
}

/* ---------------------------------------------------------- */
/* driver module load / unload                                */
/* ---------------------------------------------------------- */

static int hz=1000;                /* デフォルトの割り込み周期 (Hz)*/
MODULE_PARM(hz,"i");

static int init_module(void){
  create_proc_read_entry(DEVNAME,
                         0,                  /* デフォルト・モード */
                         NULL,                  /* 親ディレクトリ */
                         timer1_readproc,
                         NULL              /* クライアント・データ */
                         );
  init_timer1(hz);
  return 0;
}


static void cleanup_module(void){
  printk("<1>warikomi goodby¥n");
  remove_proc_entry(DEVNAME,NULL);
  ctrl_outw(0x0001, TCR1);        /* 割り込み無効, プリスケーラ =1/16 */
  free_irq(TIMER1_IRQ,NULL);
}

MODULE_LICENSE("GPL");
```

arch/sh/boot/zImageにカーネル・イメージができあがるので，CAT709の場合は実機で，

```
cp zImage /dev/mtd1
```

というようにすればカーネルを更新できます．cli()やsave_and_cli()関数などを書き換えているので，ドライバ・モジュールは再コンパイルが必要になるという点に注意してください．カーネル・コンフィグでMとしたモジュールをCAT709に導入するスマートな方法はありませんが，筆者は次のようにしています．

まず，Makefile中のINSTALL_MOD_PATH=/tmpとして/tmp以下にモジュールがインストールされるようにします．そこで，

```
# make modules_install
```

を実行すると，/tmpディレクトリに/lib/modules/2.4.21-sh-rt/ディレクトリが作られます．このファイル・ツリーを CAT709実機の/lib/modules/2.4.21-sh-rtにコピーしたのち，CAT709で，

```
# depmod -a
```

を一度実行します．

リスト1(p.157)に変更を加えたsystem.hの一部を掲載します．__cli()と__save_and_cli()に変更を加え，__real_cli()と__save_and_real_cli()を新設しました．一部の真にクリティカルな部分では，real_cli()として割り込みを本当にマスクします．

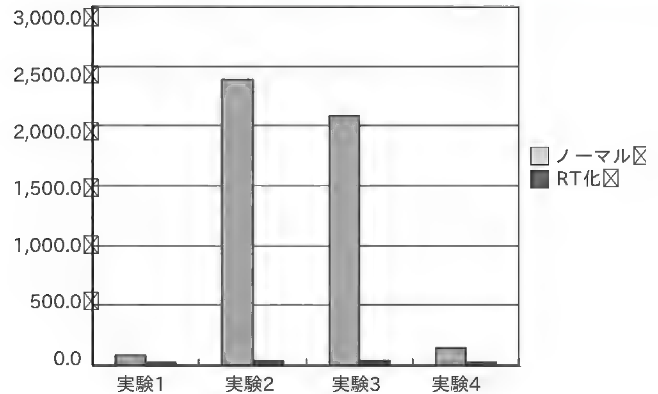
実験結果

以上のようにSH-Linuxカーネルの割り込み処理を優先度を考慮したものに変更し，割り込みレイテンシの実験を行いました．SHプロセッサにはTIMER0～2の3本のタイマがあります．このうちTIMER0は，SH-Linuxカーネルが使用していますが，TIMER1，2は未使用です．TIMER1を使用して1kHz(1msごと)に割り込みが発生するように設定し，割り込み発生から割り込みルーチンの入り口までの遅延時間を測定しました(リスト2)．測定結果を(図9)に示します．

変更前のノーマルなカーネルでは，CF(コンパクト・フラッシュ)操作時に2ms以上の割り込み遅延が発生しています．これはすなわち，ファイル操作を行っている間は，2ms以下の周期で発生している割り込みは取りこぼす可能性があることを示しています．SH-LinuxでI/O計測するときは，この数値に注意が必要です．実際，今回の実験で1kHzの割り込みはオーバランが数回発生しています．

変更後のカーネルでは平均して30μs程度の遅延時間に収まっています．オーバランも発生していません．注目すべきなのは，ファイル操作やネットワークの負荷など，Linux本体のロード・アベレージ(負荷)が上がっている状態でも割り込み処理の遅延は生じないという点でしょう．

SH-LinuxでRTLinuxやOCERAを導入することはたいへん手間がかかりますが，単に一定周期でI/O操作を行いたいだけであれば，今回改善を行ったカーネルで目的を達成できるものと考えられます．



	実験内容	ノーマル	RT化
実験1	ifconfig up	79.3	29.9
実験2	cardmgr起動	2,385.7	33.1
実験3	CFへの32Mバイトのデータ書き込み	2,090.0	32.6
実験4	ftpで8Mバイトのファイル転送	146.2	28.8

単位μs

図9　割り込み発生から割り込みルーチン到着までの遅延時間

えびはら・ゆうたろう　シリコンリナックス(株)

「VxWorks」を使ったRTOS技術の基礎と応用

第9回

組み込みシステムのデバッグ(前編)

高山 剛

本連載の第7回で，クロス開発について紹介しました．今回はさらにデバッグを焦点に，開発環境で定評のあるVxWorksで，どのようなアプローチでデバッグ環境が実現されているかを解説します．VxWorksを使っていなくても，VxWorksの組み込み特有のデバッグ・ツールの使い方や実装方法を知り，どういった場面で何ができるかを知るだけでも，現在のデバッグ環境改善の参考になると思います．また，ツールを活用した実践的なデバッグ手法も織り交ぜて紹介します．

組み込みシステムに体系立ったデバッグ手法はありません．さまざまな組み込みシステムの開発場面で，多くのエンジニアが試行錯誤の末に生み出したローテクな手法から多種多様なツールまでを紹介します．一つでも組み込み機器の開発者のデバッグに役立てばと思います．

VxWorksのデバッグ・コマンドとデバッグの方法

表1はVxWorksのコマンド一覧です．タスクの状態を変えるコマンド，タスクの状態を表示するコマンド，メモリ操作，ファイル関連，ネットワーク，シンボル操作，シングル・ステップと多種多様なコマンドをもっています．

これらのコマンドは，ターゲット・シェル(またはホスト・シェル)から呼び出すことができます．ターゲット・シェルは，VxWorks上で動作している1タスクで，標準入力から受け取った文字列を構文解析して，コマンドを実行し，結果を標準出力に出力するC言語ライクなインタプリタ(図1)です．

デフォルトでは，標準入出力はシリアル・ポート(RS-232-C)に接続されているので，ターゲットのシリアル・ポートとホストのシリアル・ポートを接続して通信します．ホスト側では通信ターミナル・ソフトウェアとして，Solarisでは`tip`コマンド，WindowsではHyperTerminalなどを使います[注1]．

こういう接続はUNIXではよく使用されていました．1台のUNIXマシンに複数のダム端末をシリアルで接続して使っていたのです．組み込み機器の開発では今も(おそらくこれからも)

注1：VGAディスプレイをコンソールにして表示する方法もあるが，スクロールで情報が失われることからデバッグに不向きなので，組み込みではVGAをもっていても，シリアル・ポートでパソコンに接続することが好まれる．

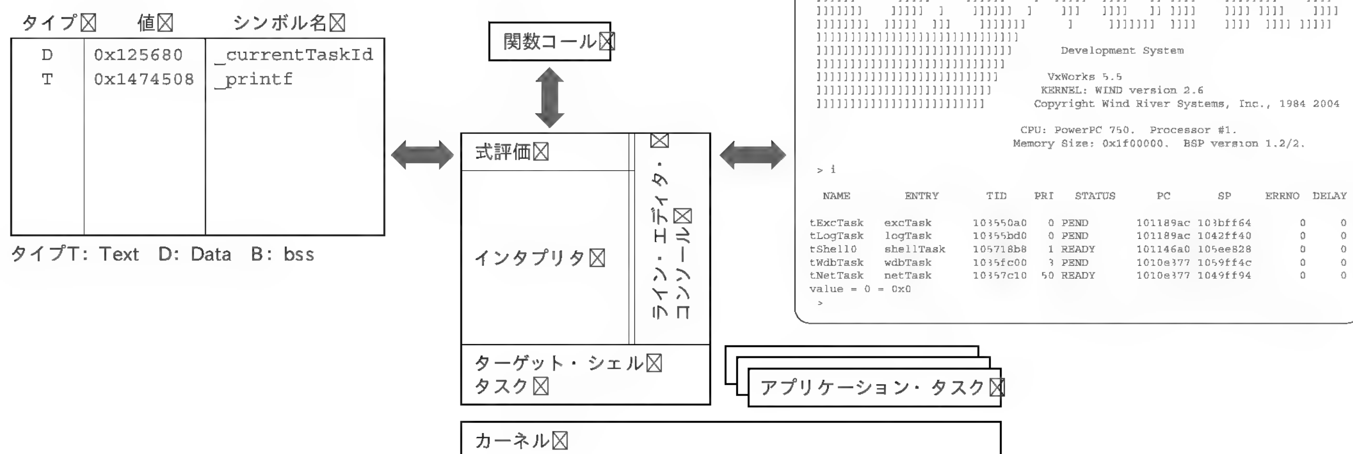


図1 シェルとシンボル・テーブル

表1 VxWorksのコマンド・ライン・ツール

h	Print(or set) shell history
i	Summary of tasks' TCBs
ti	Complete info on TCB for task
sp	Spawn a task, pri=100, opt=0, stk=20000
sps	Spawn a task, pri=100, opt=0, stk=20000
td	Delete a task
ts	Suspend a task
tr	Resume a task
tw	Print info about the object a task is pending on
w	Summary of tasks' pending information
d	Display memory
m	Modify memory
mRegs	Modify a task's registers interactively
version	Print VxWorks version info, and boot line
b	Display breakpoints/ Set breakpoint
bd	Delete breakpoint
bdall	Delete all breakpoints
c	Continue from breakpoint
s	Single step
l	List disassembled memory
tt	Do stack trace on task
bh	Set hardware breakpoint
sysSuspend	Suspend the system
sysResume	Resume the system
devs	List devices
cd/pwd	Set current working path/ Print working path
ld	Load stdin, or file, into memory
lkup	List symbols in system symbol table
lkAddr	List symbol table entries near address
printErrno	Print the name of a status value
pperio	Spawn task to call function periodically
repeat	Spawn task to call function n times
timex	Time a single execution
timexN	Time repeated executions
ls/lsr/ll	List contents of directory - long format
rename	Change name of file
copy	Copy in file to out file
cp	Copy many files to another dir
xcopy	Recursively copy files
mv	Move files into another directory
xdelete	Delete a file, wildcard list or tree
attrib/xatrib	Modify file attributes
chkdsk	Consistency check of file system
diskInit	Initialize file system on disk
diskFormat	Low level and file system disk format
e	Set eventpoint
cret	Continue to subroutine return
so	Single step/step over subroutine
spyClkStart	Start task activity monitor running
spyClkStop	Stop collecting data
spyReport	Prints display of task activity
spyStop	Stop collecting data and reports
spy	Start spyClkStart and do a report
hostAdd	Add a host to remote host table
hostShow	Print current remote host table
routeAdd	Add route to route table
routeDelete	Delete route from route table
routeShow	Print current route table
iam	Specify the user name
whoami	Print the current remote ID
rlogin/telnet	Log in to a remote host
checkStack	Print a summary of each task's stack usage
bootChange	Change the boot line
printErrno	Print the definition of a specified error status value

(a) デバッグ，ファイル・システム，ネットワーク系コマンド

taskRegsShow	Display the contents of a task's registers
taskCreateHookShow	Display hook routine
memShow	Show system memory partition blocks and statistics
envShow	Display environmental valuable
timexShow	Display the list of function calls to be timed
iosFdShow	Display a list of file descriptor names in the system
moduleShow	Show the current status for all the loaded modules
bootParamsShow	Display boot line parameters
ifShow	Show info about network interfaces
inetstatShow	Show all Internet protocol sockets
tcpstatShow	Show statistics for TCP
udpstatShow	Show statistics for UDP
ipstatShow	Show statistics for IP
icmpstatShow	Show statistics for ICMP
arptabShow	Show a list of known ARP entries
mbufShow	Show mbuf statistics
pciDeviceShow	Print information about PCI devices
dosFsShow	Display dosFs volume configuration data
scsiBlkDevShow	Show the BLK_DEV structures on a specified physical device
ataShow	Show the ATA/IDE disk parameters

(b) 主要なShowルーチン(VxWorksの各コンポーネントには，コンポーネントごとに統計情報や内部情報を報告するコマンドがmodulenameXXXShow()という名前で提供されている)

Column 1
デバッグの裏技 checksum(addr,length)

このchecksum()というのは公開されていない関数で，ネットワークのプロトコル・スタック内で使われるサブルーチンです．計算方法のアルゴリズムも，APIも公開されておらず，アプリケーションには使えませんが，デバッグには使えます．

たとえば，自身のプログラムを書き換えてしまう疑いのあるプログラムがあったとします．プログラムのあちこちに，

```
printf ("checksum %d", checksum(adr,size));
```

を埋め込んでおくと問題のあるコードの付近で値が変わるので問題箇所を発見できます．もちろん，特定のデータを含むバッファにも使えます．

使われる手法です．

デフォルトでは，シリアル・デバイスはソフトウェア・フロー制御(Xon-Xoff制御)を行っているので，大量の文字が表示されてスクロールされてしまう場合は，Ctrl＋Sキー，Ctrl＋Qキーを押すことで標準出力の停止，再開をコントロールできます．

ちなみにシリアル・マウスなど(今はもう存在しない？)を接続して，バイナリのデータを送信する場合は，ソフトウェア・フロー制御を使用しているとCtrl＋Sキー，Ctrl＋Qキーを押したことに相当する0x13，0x11を通信できず，通信が一時途絶えるように見えます．VxWorksでは，シリアル・デバイスの設定でOPT_TANDEMを選択しないことでソフトウェア・フロー制御を禁止できます．

C言語ライクでたいへん便利なターゲット・シェルの使い方

以前にも紹介しましたが，VxWorksはシステムに含まれるシンボル情報を図1の左側のように保持しているので，関数は関数呼び出し，グローバル変数はメモリの内容と評価する式評価を行います．このしくみで，VxWorksのコマンドは単なるCの関数で実現されています．したがって，bcopy()，strncpy()，strncmp()，bfill()，printf()，fopen()，read()，write()などの単なるANSIの関数さえも，コマンドとして使えます．

逆にいえばd()やi()といったコマンドとして設計されて存在するものを，プログラム中でも使用できることを意味します．プログラム中のフローのある時点でメモリ・ダンプしたい場合やタスクの状態を知りたい場合に役立つでしょう．

簡単にシェルを紹介しましょう．

▶ データ変換の例

```
->68                                             ←入力
value = 68 = 0x44 ='D'                           ←出力
->0xf5de                                         ←入力
value = 62942 = 0xf5de = _init + 0x52            ←出力
```

▶ データ計算の例

```
->(14 * 9 ) / 3                                  ←入力
value = 42 = 0x2a = '*'                          ←出力
```

▶ 変数を使用した計算の例

```
->(j + k) * 3                                    ←入力
value = …                                        ←出力
->* (j + 8 * k)                                  ←入力
value = …                                        ←出力
->x = (val1 ? val2 ) / val3                      ←入力
new symbol "x" added to symbol table             ←出力
```

▶ 式評価

シェルのC言語ライクなインタプリタは，C言語と同じ式評価が行えます．

```
->y = abs( x ) * (val1 ? val2 ) / val3           ←入力
value = …                                        ←出力
->tsk1 = taskSpawn ("dmyTask",
        10,0,1000, myTask, fd1, 300)             ←入力
value= …                                         ←出力
```

シェルのインタプリタは，式評価ルーチンで関数があれば，関数コールを行います．グローバル変数であれば参照や代入が可能です．このようにVxWorksのシェルはC言語ライクで便利なため，デバッグのみならず電卓代わりにもよく使われます．

ちょっと便利なコマンドの使い方

ここでは表1の中から，ちょっとユニークで便利なコマンドを紹介します．

● アセンブラ・レベルのブレーク・ポイントとシングル・ステップ(b，s，so，cret，c)

▶ ブレーク・ポイント設定

```
-> b printf
value = 0 = 0x0
```

▶ タスク起動とブレーク・ポイントにヒット

```
-> sp printf
Task spawned: id = 0x116f7790, name = t1
value = 292517776 = 0x116f7790
->
Break at 0x10036650: printf
Task: 0x116f7790 (t1)
```

▶ シングル・ステップ

シングル・ステップは，リスト1に示します．

▶ t1タスクのときだけ，write()関数でブレークするように設定

```
->b write, t1
```

リスト 1　シングル・ステップ

```
-> s
value = 0 = 0x0
->
edi        = 0x00000000   esi        = 0x00000000   ebp        = 0x00000000
esp        = 0x1066ffd0   ebx        = 0x00000000   edx        = 0x00000000
ecx        = 0x00000000   eax        = 0x10036650   eflags     = 0x00000246
pc         = 0x10036651   status     = 0x00000002
0x10036651  89 e5                    MOV            EBP, ESP
```

リスト 2　checkStackの使用例

```
-> checkStack
  NAME            ENTRY          TID              SIZE      CUR   HIGH    MARGIN
--------------  ------------  --------------  --------  -----  -----  -----------
tExcTask        excTask       0x102314a0        65536     156    428     65108
tLogTask        logTask       0x10232920        65536     160    272     65264
tShell0         shellTask     0x1044b7e0       131072    5220   6872    124200
tWdbTask        wdbTask       0x103d74f8        65536     180   3800     61736
tNetTask        netTask       0x10233c10        65536     108    276     65260
INTERRUPT                                       50000       0      0     50000
```

シェルより，アセンブラ・レベルでブレーク・ポイントの設定やシングル・ステップが可能です．特定のタスクだけにブレーク・ポイントを設定できるところが特徴です．これは，VxWorksのタスク・スイッチ・フックという機能を使用し，タスクのコンテクスト・スイッチが起こるたびにブレーク・ポイントを設定・削除・再設定を繰り返しているからです．

● checkStackの使用例

RTOSでは，各タスクがもつ個々のスタックは固定サイズ(タスク生成時に生成)になっています．UNIXのようにスタックがオーバフローしたときにMMUを使って自動的にページングすることはできません．したがって，スタックはタスクを生成する際にあらかじめ予測される最大値を割り当てなくてはなりません．

このcheckStack()は，そのスタック・サイズの妥当性を調べるときに役立つツールです．VxWorksはタスクを生成するとき，タスクのスタックを0xeeで埋め尽くします．checkStack()はこれを利用して，0xeeであればスタック未使用であったと仮定して，各タスクの最大使用スタック・サイズを調べます．

VxWorksのエンジニアは，ソフトウェアに問題があったとき，最初にこの機能を使ってスタック・オーバフローがあったか，余裕が十分にあるかを確認します(リスト 2)．

● lkup( )の使用例

デバッグ中，この変数，この配列，この関数のアドレスはどこだろうというときに，

```
-> lkup "symFind"
symFindSymbol            0x10092490 text   ()
```

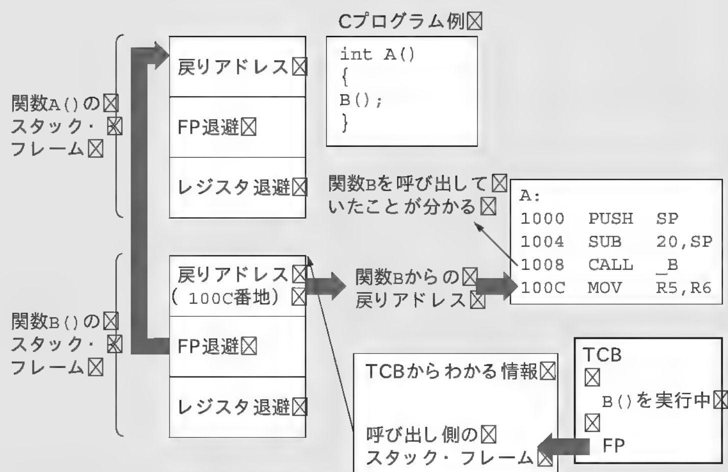
Column 2

スタック・トレースのアルゴリズム

図Aのように，スタック・トレース・コマンドtt()は，現在のTCBにあるフレーム・ポインタを使って，スタック・フレームを順々に逆トレースします．

スタック・トレース中にstatic宣言(ローカル宣言)された関数があると，そのstaticの関数ではなく，いちばん近くに存在するグローバルな関数名"xxxx"が参照されて"xxxxx+0x1022"というように，相対値を付加して表現されます．

これは，VxWorksがデフォルトではローカルなシンボルを読み込まないため起こります．正確なシンボル情報を表示したい場合は，ブート・パラメータのオプションであるflagsで0x02を設定しておくことでローカルのシンボル情報も読み込みます，スタック・トレースでローカルの関数名を正確に知りたい場合にはぜひ使用してください．



図A　スタック・トレース

```
symFindByValueAndType    0x10092990 text   ()
symFindByValue           0x10092a40 text   ()
symFindByNameAndType     0x10092770 text   ()
```

としてシンボルのアドレスを調べます．

シンボル名が長くて全部思い出せない場合は，含まれる文字をいくつか指定することで，一致するものをすべて表示してくれます．思い出せない似たようなシンボルから探し出したいときにも役立ちます．

また，VxWorksのAPIの名前規則で，システムの状態を表示するデバッグ・コマンドには，すべてXXXXShow()という名前を付けることになっているので，

```
lkup "Show"
```

とすれば，今使えるShowルーチンをすぐに見つけることができます．

● lkAddr( )の使用例

```
-> lkAddr 0x10027000
0x100263a0 fioFormatV              text
0x100270e0 vsnprintf               text
0x10027130 vsprintf                text
```

アドレスがわかっていてプログラムかグローバル変数のようだけれど，どの関数か変数かを知りたいというような場合はlkAddrが便利です．引き数で与えられたアドレス付近前後のシンボルを表示します．

```
--> l address
```

で逆アセンブルしても直前のシンボル(すなわち関数名)がわかります．

● スタック・トレースtt( )の使用例

これが最強のコマンドかもしれません．タスクがサスペンド状態，セマフォやタスク・ディレイで止まっているとき，どの関数がセマフォを呼び出しているのか，またその関数はどこから呼び出されたのかを知りたい場合にたいへん有用です．

```
--> tt タスク名(もしくはtask ID)
```

としてください．タスクのスタック上の情報を逆トレースして，タスクを立ち上げたときに最初に呼び出した関数まで，さかのぼってトレースしてくれます．

```
-> tt tWdbTask
0x100e3c93 wdbTask     +0x3  : wdbCmdLoop ()
0x100df344 wdbCmdLoop  +0x94 : wdbRpcRcv ()
0x100e0961 wdbRpcRcv   +0x31 : 0x100e41c0 ()
0x100e420c udpRcv      +0xcc : semTake ()
0x100e96ca semTake     +0xfa : 0x100e8dc6 ()
```

● printErrno( )の使用例

read()関数でファイルをリードしたけどエラーになった，ファイルはSCSIハードディスク上のファイルにあった場合，さて，ドライバでエラーなのか，SCSIライブラリなのか，それともI/Oシステムでエラーが起きたのかを見極められる方法があれば便利です．この場合は，i()コマンドでタスクの情報を表示して問題のタスクのerrnoを確認しましょう．

printfErrnoは16進数で表現される整数のerrnoを，意味ある文字列に変換してくれます．

```
-> printErrno 0x3d0001
0x3d0001 = S_objLib_OBJ_ID_ERROR
```

VxWorksのすべてのAPIは，エラーが発生すると32ビット(上位16ビットでライブラリ固有の番号，下位16ビットでライブラリごとの原因を示す)のerrno番号を設定して，関数自体はERROR(-1)を返してただちに復帰(return)します．

呼び出し側は関数呼び出しでERRORが帰ってくると，errnoを設定せず，みずからもただちに復帰します．したがって，errno番号はつねに実際の問題を起こした箇所の情報(モジュール名，原因)を示しています．

通常，errnoの設定はUNIXのプログラム・モデルと同様，慣例的に次のようなコーディングになっています．

```
errno = XXXXXX;
```

VxWorksの場合，デバッグ時を考慮して，

```
#define errno  *__errno()
```

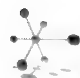
とマクロ定義されていて，__errno()は，整数型のポインタを返すしくみになっています．これにより，__errno()関数にブレーク・ポイントを設定することでerrnoを設定する箇所でつねにブレーク・ポイントにヒットできるようになっています．

真の問題を起こした直後にタスクを停止したい場合，__errno関数にブレーク・ポイントを設定してみてください．

VxWorksのAPIの呼び出しでエラーを検出した場合，とくに複数のレイヤをもっているAPIでは，errnoにより真の問題の特定が簡単にできるしくみになっています．アプリケーション側もシステムが複雑になっていくと，レイヤの構造は何階層にもなります．アプリケーション側も，このerrnoを使いこなすことで将来的にデバッグしやい，メンテナンス性の優れたコードになるでしょう．

WindRiver社のWebサイトでerrnoの一覧表を入手可能です．

https://secure.windriver.com/windsurf/search/vxWorksErrorCodes.html



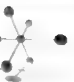
シェルを使いこなす

● VxWorksのローダ機能

VxWorksのプログラマは，プログラムを書いて，シェルを使ってプログラムをダウンロードし，プログラムをすぐに走らせることになりますが，これは次のように簡単にできます．

```
->ld < myProg.o
-> myProg
```

ここで注意しなくてはならないのは，myProg()はシェルのコンテキストで実行(さきほどの例では，シェルが構文解析して，シェルがmyProg()をサブルーチン・コールする)します．したがって，ソフトウェアの開発段階ではmyProgがコーディ



ングの問題で無限ループに陥ることもしばしばあり，この場合はシェルに復帰しません．

こういう場合でも，VxWorksのシェルはCtrl+Cキーを押すことでmyProg()を中断してシェルをリスタートできます(VxWorksのAPIの一つtaskRestart()を使っている)．

ちなみにmyProgをタスクとして起動しデバッグしたい場合は，

```
->sp myProg
```

とするだけです．

もし，myProg()にバグがあった場合，VxWorksの開発環境ではホストOS上で修正，再コンパイルを行い，ターゲットをリブートすることなく，再度myProg.oだけをダウンロードします．前回のオブジェクトはどうなるかと気になりますが，VxWorksは前回のオブジェクトを削除せず，新しいアドレス空間に新しいmyProg.oをダウンロードします．というのは，以前のオブジェクトを不用意に削除してしまうと，まだほかのタスクや自身のタスクがアクセスする可能性を秘めているからです．

シンボル・テーブルには，myProgが二重に定義されることになりますが，つねに新しいシンボルが優先されるため，古いシンボル(すなわち古いオブジェクトが使われることはない)が参照されることがないようになっています(古いシンボルの値を知りたい場合，lkupを使用する)．

メモリ資源が十分でなく，古いオブジェクトを参照して動作するタスクが存在しないことが明らかな場合は，開発者がulld()コマンドでオブジェクトを削除することも可能です．

このローダ機能は，一見危険性をはらんでいるかのように見えますが，開発者は自分のプログラムの実装，タスクの状態に熟知しているので，実際に使ってみると安全に，かつ効率よく使いこなせることに気づくでしょう．

シェルにはほかに，ヒストリ機能で，ヒストリの編集も可能です(現在はvi互換の編集機能だが，VxWorks6.0ではEmacs互換に切り替え可能)．

●リダイレクト，バッチ処理，コマンドの組み合わせ

UNIXやMS-DOS同様，リダイレクト，バッチ処理，コマンドの組み合わせも可能です．

バッチ処理やコマンドの組み合わせを紹介しましょう．

```
->d 1000 > dump
```

はリダイレクトの例です．

```
->h > batch1
->< batch1
```

リダイレクトで過去のコマンド実行の履歴をダンプし，そのファイルを編集してバッチ・ファイルとする例です(バッチ・ファイルの編集は，ホストOSのエディタを使う)．組み込みシステムの検証テストをバッチ・ファイルで実施したり，ある条件下でデバッグしたい場合，バッチ・ファイルでその条件を作り出したりするなど，バッチ・ファイルは頻繁に利用されます．

```
->a=malloc (1000)
->t1 = sp (appl1)
```

変数を利用すると，後ほどタスクやバッファをコマンドの引き数として参照する際，便利です．

```
->period 10,d,0x10000
->repeat 10,d,0x10000
```

前者は，アドレス0x10000番地を10秒置きにメモリ・ダンプ，後者は続けて10回メモリ・ダンプを行います．

```
->m "WEP64"
->d "WEP64"
```

dコマンドは，メモリをダンプするコマンドですが，これを応用した例です．無線LANのWEPキー(WEPキーは文字列でなくASCIIで指定しないといけない場合など)を電卓のような手軽さで作成できるでしょう．

```
->sp l, addr1
```

タスクを立ち上げて逆アセンブラを実行しています．意味がないように思えますが，addr1にブレーク・ポインタを設定しておいて，ブレーク・ポイントにどのようなソフトウェア・トラップを使っているのだろうと調べるときに使います．

こんなことはめったに必要でないと思われるでしょう．しかし，必要なときにコマンドを組み合わせてターゲットをコントロールできるところが，VxWorksのターゲット・シェルの最大の特徴です．

厳密にはシェルの機能ではありませんが，アプリケーション

Column 3

ISRでブレーク・ポイントにヒットしたら？

ISR(割り込みサービス・ルーチン)から関数foo()を呼び出しているとします．このとき，タスクAもfoo()を呼び出しているという状況ではどのようになるでしょう？

タスクAをシングル・ステップでデバッグしていてfoo()の中まで，ステップ・インしたとします．このとき，割り込みが発生してISRがfoo()を呼び出したらどうなるでしょうか．

ISRは確かにブレーク・ポイントにヒットしますが，OSはブレーク・ポイントにヒットした際，ISRかカーネル・ステート(スケジューラ実行中のクリティカルな操作を行っている状態)であると，ブレーク・ポイントを本来のインストラクションに置き換え，次のインストラクションにテンポラリ・ブレーク・ポイントを設定し，再実行させます．次に，もともと設定してあったブレーク・ポイントを元通り設定しなおしてから実行を再開します．

つまり，ISRでブレーク・ポイントにヒットしてもOSが命令を入れ替えるので，実際にブレーク・ポイントにヒットしているにもかかわらず，何事もなかったかのように動作するのです．

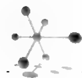
したがってISRがどのような関数を呼び出していても，デバッガで気にせず，デバッグできるわけです．

でCPU例外を起こした場合，デフォルトでOSがハンドリングを行います．CPU例外(ゼロ除算，バス・タイム・アウト，アライン例外)が起こると，問題のタスクをサスペンド状態にすることでタスクを停止させ，PC，スタック・ポインタ，ステータス・レジスタの表示，スタック・トレースの結果を表示します．

```
-> d 1234
0x000004d0:       Exception !
Vector 13 : Access Violation
Program Counter:          0x100eb22c
Access Address (read):    0x000004d2
Status Register:          0x00010246

0x100e0a19 shellTask  +0x499: shellExec ()
0x100e04e0 shellExec  +0x170: 0x100d98b0 ()
0x100d9a32 shellInterpCparse+0x11e2:
                              0x100d93fd ()
0x100d5837 shellInterpCInit+0x1297:
                              0x100d5600 ()
0x100d556d shellInterpCInit+0xfcd:
              shellInternalFunctionCall ()
0x100d19f8 shellInternalFunctionCall+0x48 :
                                       d ()
0x100eb376 d          +0x16 : 0x100eb037 ()

Shell task 'tShell0' restarted...
```



コマンドのカスタマイズ

VxWorksには，メモリ内容をエディットするコマンドとして，m()コマンドがあります．

```
->m 0x116f5458
0x116f5458:  0x0000-1234
0x116f545a:  0x0000-5678
0x116f545c:  0x0000-
```

m()コマンドは，このように元々のメモリの内容を(READして)表示して入力された新しいデータをメモリ(RAM，I/O)に書き込むのですが，I/Oをアクセスしたい場合，READされては困るI/Oもあります．こんなときは，コマンドを改造してしまいましょう．

m()コマンドは，target/src/usr/usrLib.cでソースが公開されています．

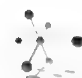
```
void m
    (
    void *adrs,     /* address to change */
    int  width      /* width of unit to be
                       modified (1, 2, 4, 8) */
    )
{
static void *lastAdrs;
                  /* last location modified */
static int  lastWidth = 2;
                /* last width - default to 2 */

if (adrs != 0)    /* set default address */
lastAdrs = adrs;

for (;; lastAdrs = (void *)
                  ((int)lastAdrs + lastWidth))
{
switch (lastWidth)
    {
    case 1:
       printf ("%08x:  %02x-", (int)
           lastAdrs, *(UINT8 *)lastAdrs);
    break;
       case 2:
       printf ("%08x:  %04x-", (int)
          lastAdrs, *(USHORT *)lastAdrs);
       break;
```

最後のswitch文の中のprintfを自分のシステムのI/Oにつごうの良いように適当に書き換えてしまいます．たとえば，アクセス・ウエイトを入れる，1バイト・アクセスでも4バイト長でアクセスする，割り込み禁止など，組み込み機器のハードウェアではそれぞれにいろいろな制限があるでしょう．このような場合は，ハードウェアに合わせてツールをカスタマイズするのが賢い方法です．

もちろん，m()をカット＆ペーストして，io()など別のコマンドとしておくほうがよいのはいうまでもありません．



ネットワークを利用したデバッグ環境の構築

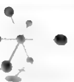
組み込みシステムがネットワーク機能を必要としない製品の場合でも，デバッグの効率化のためにネットワークを組み込み開発環境に取り入れることで，劇的な環境改善が実現できます．箇条書きで紹介すると，

▶ ネットワーク・ブートにより，電源オンから2，3秒でOSやアプリケーションが起動

ネットワーク・ブートを使わない場合，システムをリブートするまでにけっこうな時間を消費しています．組み込みシステムのデバッグではシステムをリブートする場面が多いので，リブートに要する時間も無視できません．

▶ NFSによる大容量のファイル・システム

大量のデータ，とくに画像系ではNFS経由でダンプできる



ことは有用です．原始的ですが，制御系でも`printf`で内部情報やフローを出力してNFS上のファイルにリダイレクトして事後解析する場面は多いでしょう．

▶ NFSによって，ターゲット・システムがハングアップしてもファイル・システムが破壊されない

UNIXのようにローカル・ディスクをもっていては，システム・ハングのたびにリブートして，`fsck`を実行しなければならず，リブートのたびにたいへんにむだな時間を要してしまいます．NFSを使えばターゲットをリセットしてもファイル・システムが破壊されることはありません．NFSは組み込みシステムのデバッグに使用するファイル・システムとして非常に有用です．

▶ クロス開発で快適なソース・コード・デバッグが可能

ソース・コード・デバッグには，ソース・コードとデバッグ情報をもった巨大なオブジェクト・ファイルが必要になります．クロス開発では，ホスト側にソース・コード，オブジェクト・ファイルを格納，デバッガを実行し，ターゲット側では，デバッグ・エージェトだけを動作させます．この場合，ターゲットとホスト間で通信が発生し，高度なデバッガ機能を使うと通信経路に負荷がかかる場合がありますが，ネットワークを用いれば，負荷はほとんど無視できる環境になります．

▶ 大量の情報が必要となるターゲット・システムの視覚化ツールをクロス開発環境で実現できる

たいていの組み込み機器では，グラフィック機能をもたない場合があります(まったくもたないもののほうが大多数)．このような環境でもターゲットの内部のデータや状態遷移を視覚化したい場合があり，このとき大量のデータをターゲットとホスト間で通信する必要があるのでネットワークは有用です．

▶ telnetで遠隔地から操作，メンテナンス

ネットワークを用いることにより，TCP/IPのアプリケーションのTelnet, rlogin, ftpなどをデバッグに応用できます．とくにTelnetを使えば地球の裏側の研究所から工場の特定のFA機器にインターネットを経由してデバッグできます．UNIX，Windowsから VxWorksにログインすると，VxWorksのターゲット・シェルが応答します．もちろん，TCP/IPアプリケーションなので，ほかのネットワーク・サービスであるNFSなども同時に使えます．

▶ 高価なICEが不要

▶ FTPを使ってリモート・ファイル・システムを実現可能

WindowsをホストにするNFSサーバがないときにも，VxWorksはnetDrvという機能によってFTPを使った疑似的なリモート・ファイル・システムを利用可能です．

パフォーマンスに関するデバッグ

▶ `timex()`の使い方

VxWorksはシステム・クロックをもっています．デフォルトで1 /60秒単位で周期的にタイマ割り込みが発生し，Tickという単位でカウントを行っています．`timex()`というコマンドは，指定された関数を実行して実行時間をtick単位で計測します．

```
-> timex foo
timex: time of execution =
                        183 +/- 16 (8%) millisecs
value = 53 = 0x35 = '5
```

1/60秒の単位では精度が足りない場合，`sysClkRateSet()`で周期を変えるか，次に述べる`timexN()`を使います．`timexN()`と`timex()`の違いは，計測する関数を複数回実行してクロック精度より，より高い精度の計測値を計算から求めます．

▶ `timexN()`の使い方

```
-> timexN strncpy , buf, 0x100000, 1000
timex: 125000 reps, time per rep =
                            2 +/- 0 (0%) microsecs
value = 58 = 0x3a = ':'
```

▶ `tickGet()`の使い方

`timex()`，`timexN()`の双方とも，カーネルのシステム・クロックが管理しているtickを応用しています．VxWorksのシステム・クロックは，タイマから割り込みを受けるたび，tickと

Column 4
OS awareなデバッグ

Tornado/VxWorksの開発環境では，WDBエージェントを使って最小限のデバッグ・コードをもつことでデバッグ環境を提供しています．コストの厳しいディジタル・コンシューマ機器では，当然，製品出荷時にはこれらのデバッグ用のエージェントさえも取り除いて出荷されます．

そこで，実際の製品に使われるソフトウェアをデバッグの目的のためにデバッグ・エージェントを組み込まず，ありのままのアプリケーションのバイナリをデバッグしたいという要求がつねにあります．このような場合，ICEを用いてデバッグするのですが，この場合，デバッグ・エージェントがないため，タスクの状態すら知ることが困難になります．ICEにOSの内部情報を解析させてOSをある程度認識できるようにしたデバッグを「OS awareなデバッグ」と呼びます．

VxWorksでOS awareなデバッグを行う場合，VisionICEというJTAG ICEを接続しVisionClickという専用のデバッガを使用します．Tornadoですべてできると申し分ないのですが，VisionClickがもともと他社で開発され，その後，WindRiver社が買収したという経緯のために統合化がされていませんでした． 次世代のVxWorks6.0では，このVisionClickとTornadoの技術，さらにSniffという静的C/C++コード解析ツールなどを，オープン・ソースのEclipseをIDEのフレーム・ワークとして統合化する予定になっています．

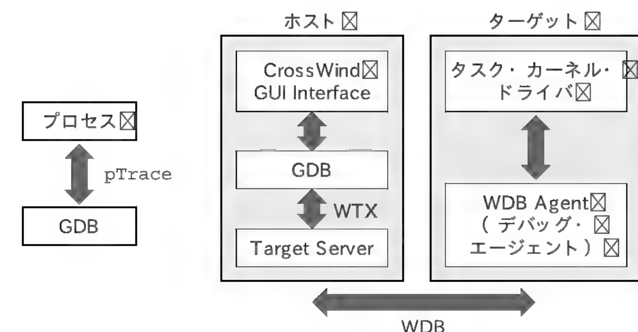


図2　GDB（GNU Debugger）の動作

呼ばれる32ビット整数のカウンタをインクリメントしています．

tickGet()で，現在のtickカウント数を参照できます．システム・クロックの周期はsysClkRateGet()で入手できるので，1 tickは実時間に換算すると，1/sysClkRateGet()秒となります．

timex()，timexN()では汎用的すぎて使用できないという場合は，アプリケーション内でtickGet()を利用するとよいでしょう．

▶ sysTimestamp()の使い方

timex()，timexN()，tickGet()より精度の高い計測方法として，WindView（システム状態遷移を捉えるシステム・アナライザ）で用いられるTimeStamp Driverを応用する方法もあります．以下に，参考のために簡易版のサンプル・コードを示します．

sysTimestamp()という関数でハードウェアのタイマ/カウンタ値を直接読みます．CPUボードの入力クロック，分周値にもよりますが，μs以下の分解能まで計測できます[注2]．

```
sysClkRateSet(10);
        /* 許される限り，タイマのインターバルを長く取る */
taskDelay( 1 );
            /* 前処理としてtaskDelay()を実行しておく，
             * 実行時間計測中のタイマのロールオーバを
             * 回避するテクニック
             */
tsStartTimer = sysTimestamp ();
your code
tsStopTimer = sysTimestamp ();
delta =( tsStopTimer - tsStartTimer );

printf ( “実行時間 %f Sec ¥n”,
     ((double) delta / sysTimestampPeriod()) /
                              sysClkRateGet() );
     /* 実行時間 =  delta / 1 tickあたりのタイマ・カウント値
                              / 1秒間あたりのtick数 */
```

▶ spy()の使い方

spy()は，各タスク，割り込みのCPU専用率を求める簡易ツールです．システム・クロック・タイマ割り込み時に，実行中のタスク（もしくは割り込み）を特定して，CPUの占有率（累積値，デルタ値）を求めます．

関数単位にパフォーマンスを知ることができないため，パフォーマンス・チューニングのためのボトルネックの検出には不向きです．この場合は次のProfileScopeツールが必要となります．

▶ WindView/ProfileScope

WindViewでタスク状態遷移，個々の割り込み，各システム・コール/ユーザ・イベント間のパフォーマンスを知ることができます．また，ProfileScope（RTI社）はさらに詳細に関数単位（関数ツリーによる）の性能計測が可能になっています．これらのツールについて，詳しくは次回のビジュアライゼーション・ツールで紹介します．

GDBでソース・コード・デバッグ

VxWorksのシェルでもアセンブラ・レベルのブレーク・ポイント，シングル・ステップは可能ですが，ソース・レベル・デバッグや構造体の表示，ローカル変数の表示まではできません．ソース・コード・レベル・デバッグを実現するため，Tornado/VxWorksの環境下では，GDB（GNUデバッガ）を採用しています．

UNIX環境下では，図2（a）のようにUNIXのプロセス制御（デバッグ）用システム・コールであるpTraceにより，GDBはアプリケーションを制御しますが，GDBはクロス環境でもpTraceを拡張して，リモートのターゲットにあるプロセス（VxWorksではタスク）をデバッグ可能です．

図2（b）のTornado/VxWorksでは，pTraceではなく，より組み込みやクロス開発を意識したWindRiver独自のWDB（Wind Debug）というプロトコルを用いています．WDBにより，タスクだけでなく，システム・モード機能によりカーネル，ドライバのデバッグが可能になるように設計されています．ホストOSとターゲット間の通信経路もDLLやshared libraryで実現されているので，DLLを選択することでシリアル，Ethernet，JTAG ICE，ROM Emulatorを経由できます．

注2：以下のコードをさらに正確にするには，sysTimestamp()自身の実行時間を計測し補正しなくてはならない．sysTimestampを繰り返し実行し，sysTimestampのおおよその実行時間を計算によって求めて補正できる．また，ハードウェアのタイマにはカウント・ダウンするもの，カウント・アップするものなどと多様な種類があり，注意が必要．

ホスト側のターゲット・サーバの役割は，GDB以外のツールと共存できるようにターゲットの情報を一元管理する目的を担っています．GDBは，Tornado下ではデバッガのエンジンとして機能し，GUIに関してはTornado開発環境に統合化されたCrossWindで実現しています．

GDBの特徴としては，あまりメモリを消費しない構造（メモリが少なかった時代に作られたため，大規模なファイルでもわざわざ分割して読み込む）で，必要に応じて必要なオブジェクト・ファイルだけのデバッグ情報をGDBに読み込むので，動作が軽快な点が挙げられます．

WDBには，Gopherという簡単なインタプリタをもっていて，非常に短いコマンド文でターゲットのメモリの複雑なデータ構造（たとえばTCB）を効率良くアクセスできるという特徴もあります．次の例は文字列1行でTCBから38個のレジスタの内容を要求するコマンド文になっています．

Gopherのコマンドと TCLを組み合わせた例（host/resource/tcl/sh-V8XX.tclより抜粋）；

```
    set regSetGopher ¥
        "<+$offset( WIND_TCB,regs)
      @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@>"
    # Comment!
           r0-r31 eipc eipsw fepc fepsw ecr psw
```

▶ CrossWind

GDBの主要なコマンドがアイコンで操作でき，すでに実行中のタスクをポップアップ・メニューに表示して後からアタッチしてデバッグが可能です．

レジスタ内容，スタック・トレース，ソース表示，ブレーク・ポイント情報，ローカル・グローバル情報などのビューも実現されています．

組み込み特有な機能強化としては，Cソース・コードを三つのモード（C言語/アセンブラ/C言語アセンブラ・ミックス）表示に切り替えることができる点が挙げられます（図3）．

システム・モードに切り替えることで，カーネル，ドライバをソース・コード・レベルでデバッグできます（しくみは後述）．

割り込みのデバッグ手法

割り込みのデバッグが困難な点は，ハードウェアと密接であること，CPUに依存すること，タイミングがクリティカルであることが挙げられます．UNIXやWindowsのアプリケーションでは，割り込みを扱うことはありませんが，RTOSではドライバだけでなくアプリケーションも割り込みを扱う場合があり，割り込みを扱いやすいようなくふうがなされています．一言で言えば，UNIXではドライバを100%コーディングして，いきなり動くか動かないかの世界ですが，VxWorksでは，ハードの初期化，割り込みが起きるか，割り込みハンドラが動くか，アクノリッジを返したか，ハードウェアのステータス・レジスタを確認…，と1段ずつステップ・バイ・ステップで確認しながら移植やデバッグができます．当然，小さな階段を少しずつ登るほうが品質が高く，結果的に速く仕事が終わることは想像できるでしょう．

まず，割り込みのデバッグに焦点を移す前にVxWorksでの割り込みについて少し解説します．

▶ VxWorksでは，割り込み処理を行うコードをC言語で記述します．これをISR（割り込みサービス・ルーチン）と呼びます．

▶ ISRは，intConnect（割り込みベクタ，ISRのアドレス，引き数）で事前に登録しておきます．

▶ 初期化コードはデバイスを初期化し，intConnect()をコールし，デバイス側の割り込み許可します．

▶ レジスタ退避などアーキテクチャに依存する部分はVxWorksのカーネルが行い，カーネルが適切なISRを呼び出します．

● logMsg()の応用

以前に紹介しましたが，割り込みをデバッグする基本は，logMsg()を使うことです．

```
    logMsg ( "message fmt" ,
      arg1,arg2,arg3,arg4,arg5,arg6,arg7,arg8);
```

logMsg()は，printf()と同じ書式を指定できる点と，同じ引き数を取るという点が似ていますが，次の点で異なります．

▶ logMsg()はISRとタスクのどちらでも呼び出し可能だが，printf()はタスクでしか使えない．

▶ logMsg()は，そのコンテキストでI/Oシステムをコールするのではなく，msgQを介してメッセージと引き数をLogTaskへ送信し，logTaskがI/Oシステムへ出力する．

ドライバの移植，動作確認をする場合は，logMsg()を使っ

図3　CrossWindのC言語とアセンブラのミックス表示

てどこまで動作したか，ISR中でI/Oのレジスタの値を表示するなどlogMsg()を呼び出すとよいでしょう．

その際，logMsg()を直接呼び出すのではなく，

```
#if 1
int xxxDrv_debug = 0;
#define XXXDRV_DEBUG(level,msg,arg1,,略) ¥
        if ( xxxDrv_debug >= level) logMsg
                              ( msg,arg1,略)
#else
#define XXXDRV_DEBUG(level,msg,arg1,,略)
#endif
```

としておくことで，VxWorksのシェルの変数への代入の機能を用いて，次のようにデバッグ情報の表示の有無，レベルをコントロールできます．

```
->xxxDrv_debug = 1
```

この方法は，VxWorksのDOSファイル・システムや各種のドライバでも実装されています．

ドライバで，このようなマクロを使うことは重要です．いったんドライバが完成したと思っても後でまた問題が生じるかもしれません．このようなデバッグ用のマクロを資産として残しておくことで，デバッグ・コードを毎回作る必要がなくなります．

● sprintf( )/memo( )の応用

logMsg()は少なくとも，シリアル・インターフェースが使用できないと使えず，シリアル・インターフェースの遅さから大量のメッセージを出力することができません．そのような場合は，OSが使用しないメモリを確保しておき，そこにsprintf()を使ってメッセージを書き込んでいく方法があります．シェルからメモリ・ダンプ，リブート後にメモリ・ダンプして事後解析が可能です．シリアル・インターフェースさえない場合には，ICEからメモリ・ダンプすれば良いでしょう．

```
sprintf ( pSprintfDebug , “ Interrupt %d ->
            %d %d”,vector , *io1, *io2);
pSprintfDebug += 32;
```

sprintf()は，I/Oシステムに関係するコードやタスクの状態を変えるシステム・コールをいっさい含んでいないので，ISRで使用できます．sprintf()では，オーバヘッドが気になるという場合(インストラクション数が数百程度で，無視できると思うが，気になってしまう方)は書式なしの次のような関数を使うとよいでしょう．

```
char * pMemo = START_MEMO;
memo( char * str , int arg1)
{
if ( pMemo > END_MEMO ) pMemo = START_MEMO;
bcopy ( str , 12 , pMemo );
*(int *)(pMemo + 12) = arg1;
pMemo += 16;
}
```

● 配列を使って統計情報を取る

Ethernetのドライバを開発していると，動作はするけど，何かの不具合で異常に遅くなるとか，ときどきTCP/IPのアプリケーションが動かなくなるなどの不具合が起こります．これはEthernetドライバの特性として受信パケットがいつ，どのタイミングで，どの程度の負荷でやってくるかわからないこと，受信バッファ，送信バッファがフルになった場合，TCP/IPのMBUFが不足するなど，異常時の処理に起因しています．要因が三つや四つだけならばフローを追うだけで済みますが，要因が十も二十にもなってくると，組み合わせ数の爆発で，フローチャートを作っても論理的に検証が難しくなります．

このような場合，何らかの要因やイベントが発生したときに配列を確保して統計情報を取っておくと各事象の相関関係や頻度を知ることができ，因果関係を類推して問題解決につながる場合があります．

```
enum {SC_NONE, SC_INIT, SC_POLL2INT,
             SC_INT2POLL}  xxxDrv[100];
   <略>
   switch (sDelta)
       {
       case SC_INT2POLL:
           if (pollTaskId == NONE)
               {
               xxxDrv[SC_INIT]++;
```

● System modeの使い方

図2(b)のように，ソース・コード・デバッガは，ターゲット側のWDB Agentによって，ターゲットのタスクの実行をコントロールしますが，Tornado/VxWorksをシステム・モードに切り替えることで，デバッグのスコープが，特定のタスクからシステム全体(カーネルとタスクを含めた)に変えることが可能です．

システム・モード下のWDB Agentは割り込み，カーネルさえ実行制御が可能となっていて，割り込み，カーネルをソース・コード・レベルでデバッグができるようになっています．どのように実現されているかを簡単に説明すると，WDB Agent内に特殊なコンテクスト(一種のTCB)をもっていて，カーネル，割り込みのコンテクスト(レジスタ，プログラム・ポインタ，スタック・ポインタ)を保持できるようになっており，カーネル，割り込みのデバッグ時は，WDB Agentがカーネルに代わって制御権をもちます．したがってWDB Agentが制御権を得ると，カーネルのスケジューラは動作しないため，マルチタスク環境は一時的に止まってしまいます[注3]．

システム・モード下のWDB Agentは，ICEのない環境でEthernet，シリアル・コネクションでも動作します．割り込み

注3：これが，システム・モードと呼ばれるゆえんである．ICEでデバッグしている感覚と同じ．

駆動で動作するEthernet，シリアルでどのようにして割り込みやカーネルのデバッグを可能にしているのかというと，VxWorksのEthernet，シリアル・ドライバは割り込み駆動のモードとポーリング・モードを両方サポートし，動的に切り替えるしくみをもっているからです．

WDB Agentは，割り込み，カーネルのデバッグで制御を得ると，割り込み禁止の状態を維持して，ドライバをポーリング・モードに切り替え，OSではなく，WDB自身が通信をすべて行います（Ethernetを使う場合は，プロトコルの単純なUDPを使う）．

OSを再開する場合は，ドライバを再度，割り込み駆動に切り替え，割り込みを許可にしてOSへ制御を返します．

● 割り込み・ドライバのデバッグのヒント

▶ 多重割り込みの発生

VxWorksは，多重割り込み（割り込み処理中，より優先順位の高い割り込みを許す）が可能です．場合によっては，ドライバがより優先順位の高い割り込みの発生を考慮せずに構築されている場合があり，クリティカル・リージョンの排他制御が不完全で問題を起こすかもしれません．

多重割り込みが発生しているかどうかを知るには，ISR中で`intCnt`というグローバル変数を参照することで多重割り込みの深さを知ることができます（`intCnt`が2，またはそれ以上の場合，多重割り込みが発生している）．

▶ シェルからISRを呼び出す．

割り込みが発生していないが，無理やりISRを呼び出すことがVxWorksでは可能です．ISRはただのC言語の関数なので，シェル（タスク・レベルで）から呼び出し実行できます．

また，シェルからハードウェアのレジスタを手作業で変更することも実際によく行われます．

▶ WorkQueueオーバフローが発生したら

VxWorksはカーネル実行中でさえ，割り込みを受け付け可能とするため，WorkQueueという機構をもっています．ISRがタスクの状態を変えるようなシステム・コールを発行した場合，直に実行するのではなく，WorkQueueにジョブを記憶し，すべてのISRが実行を終了した時点でWorkQueueのジョブを処理します．

もし，ISR中で割り込み要因に対するアクノリッジやクリアが正しくできていない場合は，再度，繰り返し割り込みが発生してWorkQueueがオーバフローすることがあります．この場合は，CPUの能力不足やWorkQueueのサイズが問題ではなく，割り込み要因に対するアクノリッジやクリアが正しくできていないことが根本的な原因です．

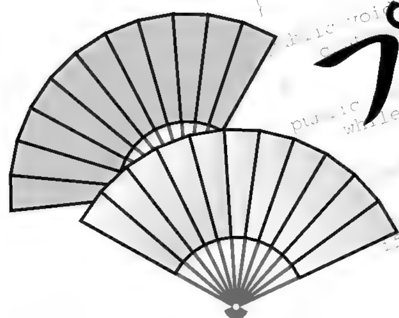
WorkQueueオーバフローが起こった場合は，割り込み要因に対するアクノリッジやクリアが行われているかどうかを確認してください．

＊　＊

次回はシミュレータでの開発，そしてWindView ScopetToolなどのビジュアライゼーション・ツールによるデバッグの手法を取り上げます．

参考文献

（1）WindRiber，VxWorks 5.5 Programmers Guide



プログラミングの要



宮坂 電人

第 17 回

ハフマン符号化による圧縮 ── 幅広く使われている圧縮アルゴリズム

圧縮

巨大なデータを取り扱うと，処理時間が長くなったりコストが高くつくことがあります．そのため，なるべく小さいサイズにデータを縮める作業，いわゆる「圧縮」を要求されることがあります．データを圧縮することで記録スペースが減り，よりたくさんのファイルが記録できたり，伝送速度の遅い通信では時間の節約になって有利です．実際，通信や情報保存で知らない間に圧縮技術のお世話になっていることがあります．

しかし，そんな便利な圧縮ですが，どうやってプログラミングすればよいかは案外知られていません．今回および次回は圧縮の原理をなるべく簡潔に紹介し，実際のプログラミング・コードも示したいと思います．

圧縮の分類

圧縮とは「元のデータをサイズが小さくなるように特別な『符号』に変換する」作業といえます．この作業は「符号化」とも称します．一方，「符号に置き換えられた情報を元の情報に復元する」作業を「展開」と称したり「解凍」と称します．この作業は「復号化」とも称します．一般的には圧縮した情報を復号化すると必ず元の情報が復元されること（可逆圧縮）を期待しますが，圧縮率が高い代償として完全に復元できない圧縮法（不可逆圧縮）もあります．今回と次回紹介する圧縮法はいずれも可逆圧縮です．

可逆圧縮には，処理の最初から最後まで元データに対応する符号が一定である「静的符号化（static coding）」と，処理が進むにつれ元データに対応する符号が変化する「動的符号化（dynamic coding）」があります．今回紹介するランレングス符号化とハフマン符号化は静的符号化に分類されるもので，次回紹介する予定のLZ符号化は動的符号化に分類されます（図1）．

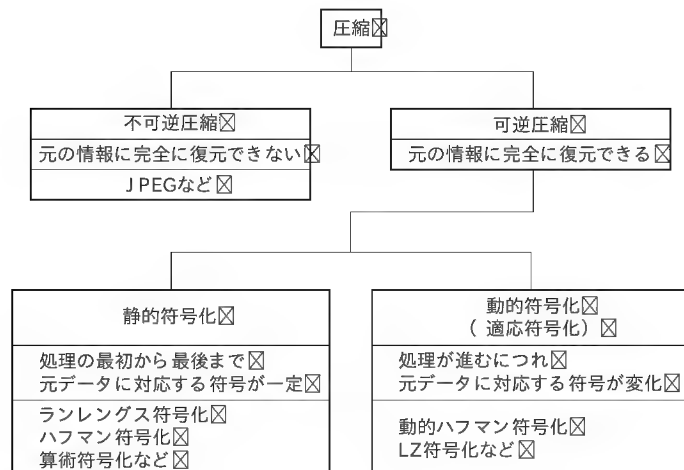


図1　圧縮方法の分類

情報の偏りとランレングス符号化

圧縮の原理を知らない人にとって圧縮とは何やら途方もなく難しい原理を応用しているように思えます．しかし簡単にいい切ってしまえば，

● 情報の偏りに着目し，それを利用して符号化する

ことが圧縮の原理です．もっとも単純な「ランレングス符号化（run-length encoding：RLE）」で説明しましょう．画像情報を例にするとわかりやすいと思いますが，まったく同じ値のデータが連続して多数並ぶことがあります．たとえば，「AAAAABBBCCCCCCC」という15バイトの情報があったとします．見てわかるとおりAというシンボルが連続して5回，Bが3回，Cが7回出現しています．ここで出現回数をそれぞれのシンボルの前につけ，シンボルが重複して出現しないよう符号化すると「5A3B7C」となります．つまり15バイトあった元情報が6バイトに圧縮されたわけです．このようにシンボルの連続出現回数を付加することで圧縮を試みる符号化がランレングス符号化です．ランレングス符号化は原理がわかりやすく，圧縮や展開のプログラムを作ることが簡単なので実装する機会も多いでしょう．

ところで，さきほどの圧縮の原理で「情報の偏りに着目」とあ

りましたが，ランレングス符号化が着目した情報の偏りとは「同じシンボルが連続して配置されることがある」です．「それを利用して符号化」とは「連続出現回数を付加することで同じシンボルを連続して配置しなくていい」ということです．

しかしながら同じシンボルが連続して配置されない状況だと，とたんに不利になります．たとえば「TICKTACKTICKTOE」という，やはり15バイトの情報があった場合，これをランレングス符号化すると「1T1I1C1K1T1A1C1K1T1I1C1K1T1O1E」で30バイトに増えてしまい圧縮にはなりません．いうまでもないことですが「TICKTACKTICKTOE」には連続して出現するシンボルがないのですから，これでは「利用」しようがありません．

シンボルの対応ビット数を減らす

ランレングス符号化で圧縮できなかった「TICKTACKTICKTOE」ですが，よく見ると出現しているシンボルは「A，C，E，I，K，O，T」の7種類しかありません．ということは一つのシンボルに8ビット（1バイト）も割り当てる必要がありません．3ビットあれば十分です[注1]．ということで，

A＝000，C＝001，E＝010，I＝011，

K＝100，O＝101，T＝110

という2進数に置き換える符号化をすると「110011001100110000001100110011001100110101010」となり，45ビットになります．元の情報が120ビット（8ビット×15）だったので，これはけっこう圧縮率が良いように思います．しかし，この方法の限界は情報量が増えると，結局一つのシンボルに8ビットを割り当てざるをえない状況が出てきやすい点です．

ハフマン符号化（Huffman coding）

さらに「利用」できる「情報の偏り」がないかを検討してみると「TICKTACKTICKTOE」で使用するシンボル，すなわち「A，C，E，I，K，O，T」の出現頻度にはばらつきがあります．つまり，

A＝1回，C＝3回，E＝1回，I＝2回，

K＝3回，O＝1回，T＝4回

で，Tの出現頻度がもっとも高く，A，E，I，Oは1回しかありません．ここで出現頻度の高いシンボルに少ないビット数の符号を割り当て，出現頻度の低いシンボルに多いビット数の符号を割り当ててみます．たとえば，

A＝1001，C＝000，E＝1000，I＝001，

K＝11，　O＝101，T＝01

に置き換えてみると「0100100011011001000110100100011011011000」となり，40ビットに圧縮されます．さきほどのランレングス符号化とは違い，同じデータが連続しているかに着目しているのではなく，シンボルの出現頻度に着目しているため，データの連続が少なくても圧縮しやすいのが，この符号化の特徴です．ここで示した圧縮方法が「ハフマン符号化」と呼ばれるものです[注2]．ハフマン符号化の特徴は，一つのシンボルを固定長のビット数の符号に変換するのではなく，可変長のビット数の符号に変換する点です．そして出現頻度の高いシンボルになるべく少ないビット数の符号を割り当て，出現頻度の低いシンボルに多いビット数の符号を割り当てることで，全体のサイズを縮めてしまうという，わかってしまえば，なんとも単純な原理です．しかし，こんな単純な原理が20世紀の半ばに至るまでだれも思いつかなかったのですから，そちらのほうが驚くべき事実かもしれません．

ところで符号の割り当てですが，さきほどの割り当てビット数は多いではないかと考える読者もいるでしょう．たとえば，

A＝110，C＝11，　E＝101，I＝111，

K＝01，　O＝100，T＝0

で置き換えれば「0111110101101101011111010100101」で31ビットに圧縮できます．しかし，これは復元できません．先頭の「0111」が「TI」なのか「KC」なのか判別できないからです．きちんと復元できるためには「符号木」に登録できる条件を満たす必要があります[注3]．符号木とは符号情報を解読できるツリーで次のような性質があります．

（1）ツリーにあるノードはリーフ（子ノードがまったくないノード）または子ノードを二つ所持するノードのいずれかで，子ノードを一つしか所持しないノードは存在しない

（2）リーフには復元すべきシンボル情報が記録されている

さきほどの「A，C，E，I，K，O，T」を元に作成された符号木を利用すると「0111」は「TK」にしか復元できないのが確認できるでしょう（図2）．

符号木の構築

符号木に登録するシンボルは出現頻度が高いものほどルート・ノードから近いリーフにあり，出現頻度が低いものほどルート・ノードから遠いリーフにあります．問題はどのように符号木を構築するかです．これは以下のような手順で実現できます．

- ノードに「シンボル」，「出現頻度」を記録できるようにする

リスト1に示すのは，符号木に登録するノードのプログラム例です．

2種類のコンストラクタがありますが，それぞれリーフ作成

注1：$2^3=8$なので，3ビットあれば8種類まで対応できる．

注2：ハフマンはこの圧縮法を発表した人の名前．David A. Huffman（1925～1999）．1952年，彼が大学院生だったときに発表した．彼の履歴については http://en.wikipedia.org/wiki/David_A._Huffman を参照．

注3：ハフマン符号化の処理過程で生成される符号木を「ハフマン・ツリー」あるいは「ハフマン木」とも称する．

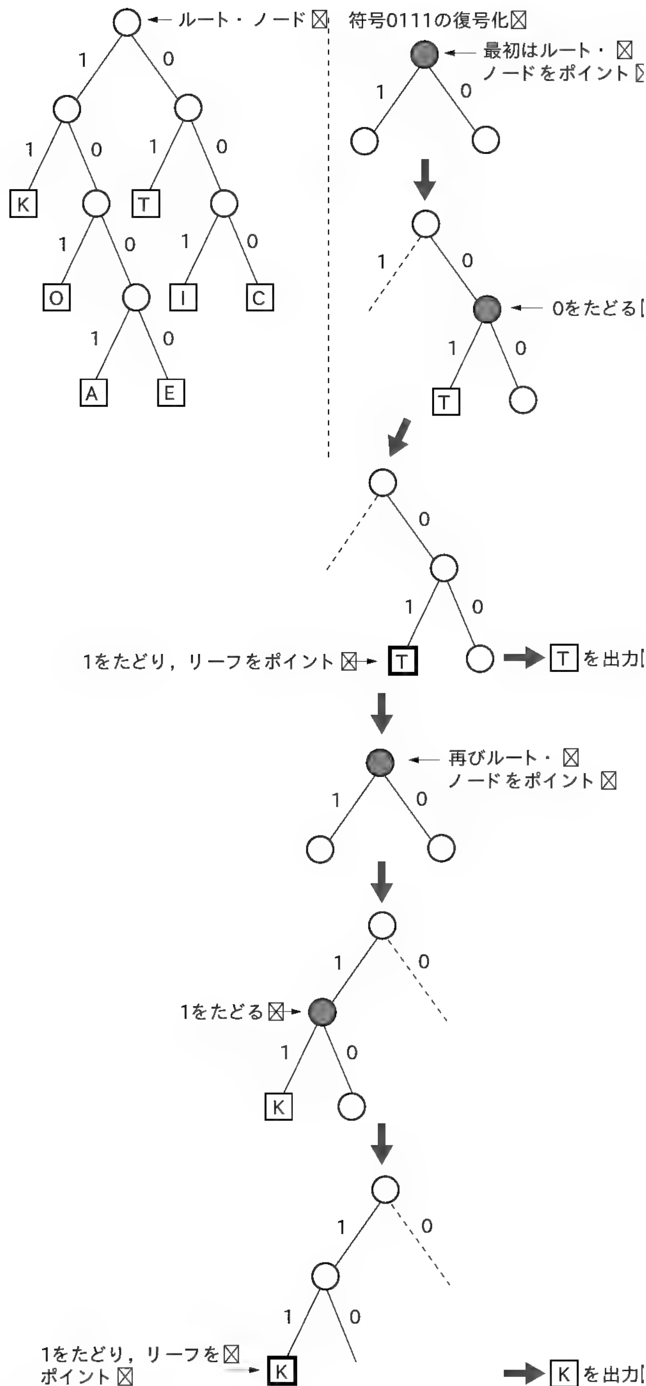


図2　符号木

用と，二つの子ノードを所持するノードの作成用です．

- 圧縮したい情報を全走査し，シンボルの出現頻度テーブルを作成する
- 出現頻度テーブルを元にリーフ・ノードを作成し，それらを優先度つきキューに登録する

ここで優先度といっているのは，出現頻度が低いノードを優先してキューの先頭から取り出せる状況のことです．

リスト1　HuffTreeNodeクラス

```
//符号木のノード
struct HuffTreeNode {
    uint_32_t freq;              //出現頻度
    HuffTreeNode* littleFreq;    //出現頻度の低い
                                 // 子ノードへのポインタ
    HuffTreeNode* bigFreq;       //出現頻度の高い
                                 // 子ノードへのポインタ
    uint_8_t symbol;             //文字コード

    //リーフ・ノードの新規作成用コンストラクタ
    HuffTreeNode(uint_8_t iSymbol,uint_32_t iFreq) {
        symbol = iSymbol;
        freq = iFreq;
        bigFreq = littleFreq = NULL;
    }
    //二つのノードを子にするノードの新規作成用コンストラクタ
    //i1=出現頻度が低い側のノード,i2=出現頻度がi1と等しいか,
    // より高い側のノード
    HuffTreeNode(HuffTreeNode* i1,HuffTreeNode* i2) {
        //子ノードの出現頻度の合計を自分の出現頻度とする
        freq = i1->freq + i2->freq;
        littleFreq = i1;
        bigFreq = i2;
    }
    //デストラクタ
    ~HuffTreeNode() {
        //子ノードを再帰的に解放する
        delete bigFreq;
        delete littleFreq;
    }
    //リーフ(自分に子ノードがない)かどうかの確認
    bool is_leaf() const {
        return (bigFreq == NULL && littleFreq == NULL);
    }
};
```

- 優先度つきキューから二つのノードを取り出し，この二つを子ノードにする新たなノードを作成し，それを優先度つきキューに登録する

この操作を優先度つきキューから取り出せるノードが一つになるまで繰り返します(図3)．以上述べた手順をプログラムするとリスト2(p.176)のようになります．

ビット単位の入出力対策

ハフマン符号化と復号化の本体部分を説明したいところですが，その前に片付けておくべき課題があります．というのは，可変長ビットの情報を読み書きするため，その処理(ビット操作やローテーション処理など)をそのままプログラム・コードで記述すると第三者にとって極めて読みにくいものになります．本番で利用するプログラムなら実行効率の問題もあって，あえてそうせざるをえない場合が多々あるでしょう．しかし本連載で紹介するコードとしては，読者が理解しにくいコードになるため不都合です．また符号化も復号化も，どの入出力装置に対してでも起こりうる処理なので，メモリ上で処理すると限定するのは不都合でしょう(同様にファイル処理だけと限定するのも)．エンディアン問題も考慮しておくべきです．ということで以下のようなクラスを作成し，符号化と復号化はそれぞれのクラスを介して行うようにします．

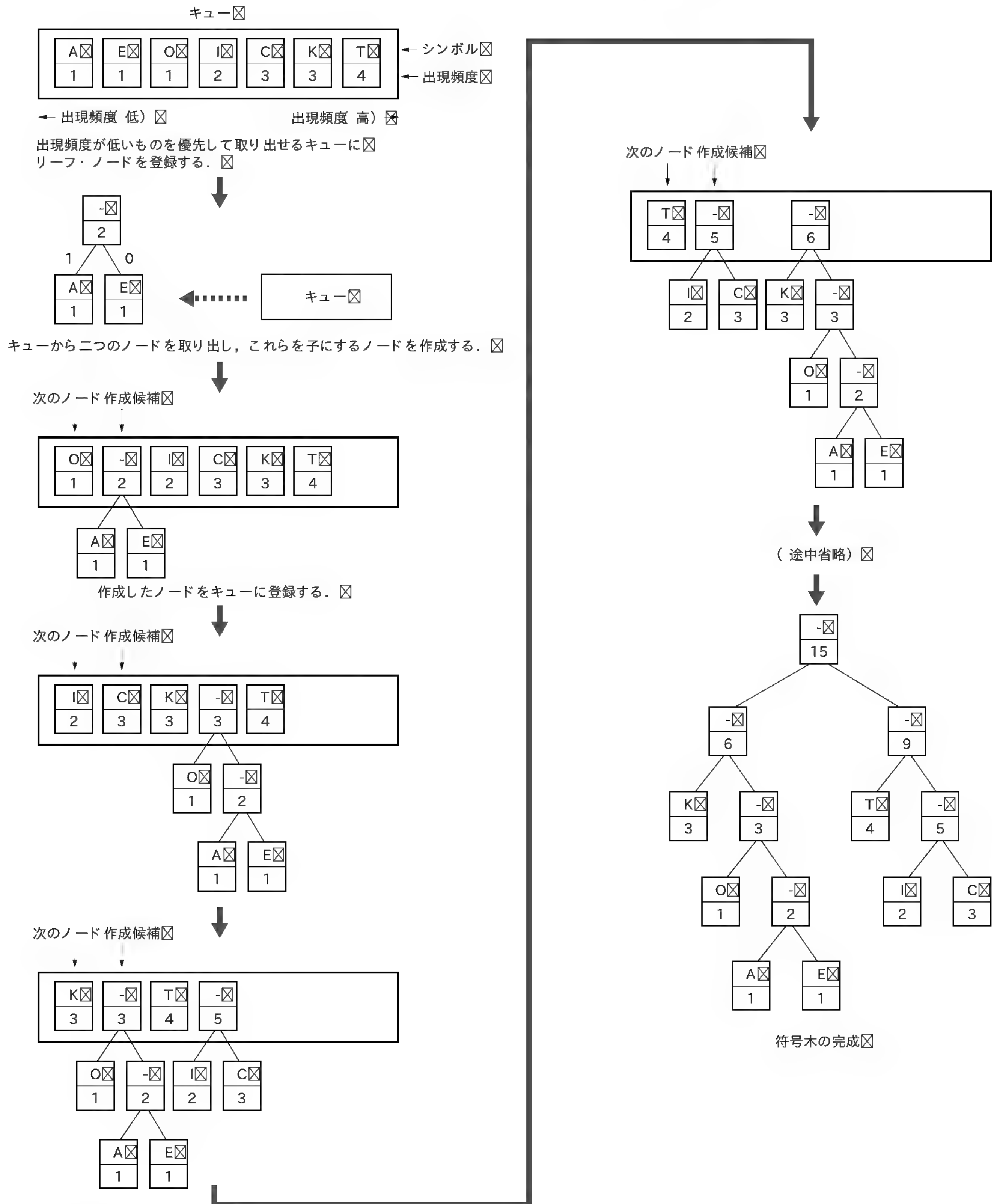

キュー
シンボル
出現頻度
出現頻度（低）
出現頻度（高）
出現頻度が低いものを優先して取り出せるキューに
リーフ・ノードを登録する．
キュー
キューから二つのノードを取り出し，これらを子にするノードを作成する．
次のノード作成候補
作成したノードをキューに登録する．
次のノード作成候補
次のノード作成候補
次のノード作成候補
（途中省略）
符号木の完成

図3　符号木の構築

リスト 2　HuffCompressクラス（符号木を組み立てる部分）

```
typedef HuffTreeNode* HuffTreeNodePtr;

//出現頻度テーブル，ハフマン・コード対応表のサイズ
#define HC_ApTableSize 0x100

//優先度つきキューでの比較用関数オブジェクト
struct HuffTreeNodeCompare : public std::
  binary_function<HuffTreeNodePtr,HuffTreeNodePtr,bool> {
    bool operator()(const HuffTreeNodePtr& i1,
                    const HuffTreeNodePtr& i2) const {
        return i1->freq > i2->freq;
    }
};

//ハフマン圧縮のクラス
class HuffCompress {
    uint_32_t mApTable[HC_ApTableSize]; //出現頻度テーブル
    //符号木のルート・ノードへのポインタ
    HuffTreeNodePtr mHuffTreeRoot;
    ...(略)...
    //出現頻度テーブルから符号木を組み立てる，成功すればtrue
    bool build_huff_tree() {
        //すでに構築した符号木を解放する
        delete mHuffTreeRoot;
        mHuffTreeRoot = NULL;
        //ツリー・ノードの優先度つきキューを用意する
        std::priority_queue<HuffTreeNodePtr,std::
          vector<HuffTreeNodePtr>,HuffTreeNodeCompare>
            aNodeQueue;
        //符号木構築の準備
        for(int aI = 0; aI < HC_ApTableSize; aI++){
            uint_32_t aFreq = mApTable[aI];
            //出現頻度のあるもののみキューに登録する
            if(aFreq > 0){
                aNodeQueue.push(new HuffTreeNode(aI,aFreq));
            }
        }
        //キューの登録ノードがない場合，エラーと判断して戻る
        if(aNodeQueue.size() == 0){
            return false;
        }
        //キューの登録ノードが1つを越える限り，キューの
        // 先頭と，その次のノードを子にするノードを作り，
        // それをキューに登録する
        while(aNodeQueue.size() > 1){
            HuffTreeNodePtr aNode1 = aNodeQueue.top();
            aNodeQueue.pop();
            HuffTreeNodePtr aNode2 = aNodeQueue.top();
            aNodeQueue.pop();
            aNodeQueue.push(new HuffTreeNode(aNode1,aNode2));
        }
        //キューの先頭にあるのが符号木のルート・ノード
        mHuffTreeRoot = aNodeQueue.top();
        return true;
    }
```

リスト 3　ReadByteBaseクラス

```
// 型の定義 （コンパイラによって書き換えるべき定数）
typedef signed char    int_8_t;   // 8ビット符号あり整数の型
typedef signed short   int_16_t;  //16ビット符号あり整数の型
typedef signed long    int_32_t;  //32ビット符号あり整数の型
typedef unsigned char  uint_8_t;  // 8ビット無符号整数の型
typedef unsigned short uint_16_t; //16ビット無符号整数の型
typedef unsigned long  uint_32_t; //32ビット無符号整数の型

class ReadByteBase {
public:
    //1バイトを読み取る，成功すればtrue，読み取り
    // 結果はoByteに置く
    virtual bool read_1byte(uint_8_t& oByte) = 0;
    //読み取り位置を先頭からiPosバイト目(0～)にする
    // 成功すればtrue
    virtual bool rewind_ptr(uint_32_t iPos) = 0;

    //2バイトを読み取る，成功すればtrue，読み取り
    // 結果はoWordに置く
    bool read_2byte(uint_16_t& oWord){
        uint_8_t a1,a2;
        if(read_1byte(a1) && read_1byte(a2)){
            oWord = (a1 << 8) | a2;
            return true;
        }
        return false;
    }
    //4バイトを読み取る，成功すればtrue，読み取り
    // 結果はoDwordに置く
    bool read_4byte(uint_32_t& oDword){
        uint_16_t a1,a2;
        if(read_2byte(a1) && read_2byte(a2)){
            oDword = (a1 << 16) | a2;
            return true;
        }
        return false;
    }
};
```

● ReadByteBase ――バイト単位で読み取りをするオブジェクトのベース・クラス

リスト3のようなクラスを用意します．読み取りオブジェクト（ファイル，ネット通信，メモリ，その他のいずれでも）はこのクラスを継承し，read_1byte（1バイトの読み取り）とrewind_ptr（読み取り開始位置の変更）の両メンバ関数を用意する必要があります．

このクラスを継承した，ファイルの読み取りクラスはリスト4のようになります．

● WriteByteBase ――バイト単位で書き込みをするオブジェクトのベース・クラス

リスト5のようなクラスを用意します．書き込みオブジェクトはこのクラスを継承し，write_1byte（1バイトの書き込み）とclose_buffer（書き込み終了）の両メンバ関数を用意する必要があります．

このクラスを継承した，ファイルの書き込みクラスはリスト6のようになります．

● ReadBitClass ――ビット単位で読み取りするクラス

バイト単位で読み取るクラス（ReadByteBase）にアダプタとして付着し，ビット単位で読み取る機能を付加するクラスです．リスト7（p.178）のようになります．

● WriteBitClass ――ビット単位で書き込むクラス

バイト単位で書き込むクラス（WriteByteBase）にアダプタとして付着し，ビット単位で書き込む機能を付加するクラスです．リスト8（p.178）のようになります．

ハフマン符号化の符号化処理部分

ハフマン符号化処理は，単純に考えると以下のような手順となります．

リスト 4　ReadByteFileクラス

```
class ReadByteFile : public ReadByteBase {
    std::FILE* mFp;
public:
    ReadByteFile(){
        mFp = NULL;
    }
    virtual ~ReadByteFile(){
        close_file();
    }
    //ファイルを開く,成功すればtrue
    bool open_file(const char* iFileName){
        close_file(); //(念のため)
        mFp = std::fopen(iFileName,"rb");
        return (mFp != NULL);
    }
    //ファイルを閉じる,成功すればtrue
    bool close_file(){
        bool aResult = true;
        if(mFp != NULL){
            aResult = (std::fclose(mFp) == 0);
            mFp = NULL;
        }
        return aResult;
    }
    //1バイトを読み取る,成功すればtrue,読み取り
    // 結果はoByteに置く
    bool read_1byte(uint_8_t& oByte){
        int aDat = std::fgetc(mFp);
        if(aDat == EOF){
            return false;
        }else{
            oByte = static_cast<uint_8_t>(aDat);
            return true;
        }
    }
    //読み取り位置を先頭からiPosバイト目(0～)にする
    // 成功すればtrue
    bool rewind_ptr(uint_32_t iPos){
        return (std::fseek(mFp,iPos,SEEK_SET) == 0);
    }
    //ファイル・サイズをoFileSizeに書き込む,成功すればtrue
    bool get_file_size(uint_32_t& oFileSize){
        bool aResult = false;
        long aCurPos = std::ftell(mFp);
        if(aCurPos >= 0){
            if(std::fseek(mFp,0,SEEK_END) == 0){
                long aTailPos = std::ftell(mFp);
                if(aTailPos >= 0){
                    oFileSize = static_cast<uint_32_t>(
                                                    aTailPos);
                    aResult = true;
                }
            }
            std::fseek(mFp,aCurPos,SEEK_SET);
        }
        return aResult;
    }
};
```

リスト 5　WriteByteBaseクラス

```
class WriteByteBase {
    uint_32_t mWroteSize; //書き込みに成功したサイズ
public:
    WriteByteBase(){
        reset_wrote_size();
    }
    //書き込み成功サイズをリセットする
    void reset_wrote_size() {
        mWroteSize = 0;
    }
    //書き込み成功サイズをえる
    uint_32_t get_wrote_size() const {
        return mWroteSize;
    }
    //書き込みの終了をする(バッファのflushも含めて)
    // 成功すればtrue
    virtual bool close_buffer() = 0;

    //1バイトを書き込む,成功すればtrue,iByteが書き込みデータ
    /*
        このメンバ関数は継承した側で必ずオーバライドする
         必要がある.
        そのさい↓のようなオーバライドを必ず行うこと
        virtual bool write_1byte(uint_8_t iByte){
            ...(継承側の独自の処理)...
            if(処理が成功){
                return WriteByteBase::write_1byte(iByte);
            }else{
                return false;
            }
        }
    */
    virtual bool write_1byte(uint_8_t iByte){
        ++mWroteSize;
        return true;
    }
    //2バイトを書き込む,成功すればtrue,iWordが書き込みデータ
    bool write_2byte(uint_16_t iWord){
        return (write_1byte((iWord >> 8) & 0xff)
                && write_1byte(iWord & 0xff));
    }
    //4バイトを書き込む,成功すればtrue,iDwordが書き込みデータ
    bool write_4byte(uint_32_t iDword){
        return (write_2byte((iDword >> 16) & 0xffff)
                && write_2byte(iDword & 0xffff));
    }
};
```

リスト 6　WriteByteFileクラス

```
class WriteByteFile : public WriteByteBase {
    std::FILE* mFp;
public:
    WriteByteFile(){
        mFp = NULL;
    }
    virtual ~WriteByteFile(){
        close_file();
    }
    //ファイルを開く,成功すればtrue
    bool open_file(const char* iFileName){
        close_file(); //(念のため)
        mFp = std::fopen(iFileName,"wb");
        return (mFp != NULL);
    }
    //ファイルを閉じる,成功すればtrue
    bool close_file(){
        bool aResult = true;
        if(mFp != NULL){
            aResult = (std::fclose(mFp) == 0);
            mFp = NULL;
        }
        return aResult;
    }
    bool close_buffer(){
        return close_file();
    }
    //1バイトを書き込む,成功すればtrue,iByteが書き込みデータ
    bool write_1byte(uint_8_t iByte) {
        int aDat = std::fputc(iByte,mFp);
        return (aDat != EOF) ? WriteByteBase::write_1byte(
                                              iByte) : false;
    }
};
```

リスト 7　ReadBitClass クラス

```
class ReadBitClass {
    ReadByteBase& mRBB; //バイト単位で読み取るオブジェクト
    uint_8_t mReadData; //読み込んだ1バイト分の一時保管
    //1ビットを取り出すためのマスク(0x80～0x00,0x00なら
    // mReadDataは無効とみなす)
    uint_8_t mMaskByte;

    ReadBitClass(); //(empty)
public:
    ReadBitClass(ReadByteBase& iRBB) : mRBB(iRBB) {
        //まだ1バイトも読み込んでいない状態にする
        mMaskByte = 0;
    }
    //1ビットを読み取る,成功すればtrue
    // 読み取り結果はoByteに置く
    bool read_1bit(uint_8_t& oByte) {
        if(mMaskByte == 0){ //まだ1バイトも読んでいないなら
                            //1バイト分を読み取る
            if(!mRBB.read_1byte(mReadData)){
                return false;
            }
            mMaskByte = 0x80; //マスクを最上位ビットにする
        }
        //読み取り済みのデータとマスクから0,1をえて
        // oByteに反映する
        oByte = (mMaskByte & mReadData) ? 1 : 0;
        //次のマスク・パターンに変更する
        mMaskByte >>= 1;
        return true;
    }
    //複数ビットを読み取る,成功すればtrue
    // 読み取り結果はoDwordに置く
    //iBitsは読み取るビット数(1～32)
    bool read_bits(int iBits,uint_32_t& oDword) {
        uint_32_t aDword = 0;
        for(int aI = 0; aI < iBits; aI++){
            uint_8_t aBit;
            if(!read_1bit(aBit)){
                return false;
            }
            aDword = (aDword << 1) | aBit;
        }
        oDword = aDword;
        return true;
    }
};
```

リスト 8　WriteBitClass クラス

```
class WriteBitClass {
    WriteByteBase& mWBB; //バイト単位で書き込むオブジェクト
    uint_8_t mWriteData; //1バイト分になるまで一時保管
    //1ビットを記録するためのマスク(0x80～0x00,0x00なら
    // mWriteDataは無効とみなす)
    uint_8_t mMaskByte;

    WriteBitClass(); //(empty)
public:
    WriteBitClass(WriteByteBase& iWBB) : mWBB(iWBB) {
        //まだ書き込むべきデータがない状態にする
        mMaskByte = 0;
    }
    //書き込みを終了する(バッファのflushも含めて)
    // 成功すればtrue
    bool close_buffer() {
        bool aResult = true;
        if(mMaskByte != 0){
            aResult = mWBB.write_1byte(mWriteData);
            mMaskByte = 0;
        }
        if(!mWBB.close_buffer()){
            aResult = false;
        }
        return aResult;
    }
    //1ビットを書き込む,成功すればtrue
    // iDatは書き込みデータ(0または1)
    bool write_1bit(int iDat) {
        if(mMaskByte == 0){    //一時保管しているデータがないなら
            mWriteData = 0;    //一時保管場所をクリアし
            mMaskByte = 0x80; //マスク・パターンを初期化する
        }
        if(iDat != 0){ //書き込みデータが1なら
            mWriteData |= mMaskByte; //マスク・パターンを埋める
        }
        mMaskByte >>= 1;     //次のマスク・パターンを用意する
        if(mMaskByte == 0){ //全ビットを走査したなら
                             //できあがった1バイトを書き込む
            return mWBB.write_1byte(mWriteData);
        }else{
            return true;
        }
    }
    //複ビット数を書き込む,成功すればtrue
    //iBitsは書き込むビット数(1～32),iDatは書き込みデータ
    bool write_bits(int iBits,uint_32_t iDat) {
        uint_32_t aMaskPat = 1 << (iBits - 1);
        for(int aI = 0; aI < iBits; aI++){
            if(!write_1bit((aMaskPat & iDat) ? 1 : 0)){
                return false;
            }
            aMaskPat >>= 1;
        }
        return true;
    }
};
```

(1) 圧縮したい情報を全走査し，シンボルの出現頻度テーブルを作成する
(2) 出現頻度テーブルから符号木を組み立てる
(3) 符号木からハフマン・コード対応表を作成する
(4) 対応表を元に圧縮したい情報の全シンボルを符号に変換していく

ハフマン・コード対応表が必要なのは，元データを符号化するときに符号木をそのまま利用しにくいからです．符号木はその名前とはうらはらに，復号には便利でも符号を作るのには便利ではありません．対応表とは具体的には元データのシンボルに対して，どのような符号を割り当てるかを示す配列です．対応表を作成する部分はリスト 9のようになります．

ところで，ハフマン符号化したとき気になるのは，符号木をどう記録するかです．そのままを書き込むわけにはいかないので，何らかのシリアライズ処理[注4]が必要です．しかし，符号木の記録で圧縮データのサイズが増えては本末転倒です．よく考えれば符号木は出現頻度テーブルから作成できるので，代わりに出現頻度テーブルを記録してもかまいません．復号化のときは記録していた出現頻度テーブルを取り出し，そこから符号木を復元すればよいのです．ただ，出現頻度テーブルは1024バイト(4バイト×256だから)もあり，これも圧縮という目的に

注4：オブジェクトを後から復元できるような配慮をほどこして記録する処理．

リスト 9　HuffCompressクラス（ハフマン・コード対応表作成部分）

```
    //ハフマン・コード対応表となる配列
    struct {
        uint_32_t data; //符号データ
        uint_32_t len;  //符号データの長さ（0なら対応符号なし）
    } mHuffCodeTable[HC_ApTableSize];
    ...(略)...
    //符号木からハフマン・コード対応表を作成する
    void build_huff_code_table() {
        //ハフマン・コード対応表をクリアする
        std::memset(mHuffCodeTable,0,sizeof(mHuffCodeTable));
        //再帰的にツリーをたどり対応表を作成する
        build_hc_table_sub(mHuffTreeRoot,0,0);
    }
    //build_huff_code_tableの下請け
    //iHTN=ノード,iData=符号データ,iLen=符号データの長さ
    void build_hc_table_sub(HuffTreeNodePtr iHTN,
                        uint_32_t iData,uint_32_t iLen) {
        if(iHTN->is_leaf()){ //リーフである場合
            //ルート・ノードがリーフであったときの対策
            if(iLen == 0){
                iLen = 1;
            }
            uint_8_t aSymbol = iHTN->symbol;
            mHuffCodeTable[aSymbol].data = iData;
            mHuffCodeTable[aSymbol].len = iLen;
        }else{ //リーフでない場合
            ++iLen;
            iData <<= 1;
            //出現頻度の高い側のノードをたどる （値の解釈は0）
            build_hc_table_sub(iHTN->bigFreq,iData,iLen);
            //出現頻度の低い側のノードをたどる （値の解釈は1）
            build_hc_table_sub(iHTN->littleFreq,iData | 1,
                                iLen);
        }
    }
```

リスト 10　HuffCompressクラス（出現頻度を縮小する処理）

```
    //出現頻度テーブルの値を0～255の範囲内におさめる
    void shrink_ap_table() {
        int aI;
        //出現頻度テーブルの最大値を求める
        uint_32_t aMax = 0;
        for(aI = 0; aI < HC_ApTableSize; aI++){
            if(mApTable[aI] > aMax){
                aMax = mApTable[aI];
            }
        }
        //最大値が255以内におさまるなら調整不要なので戻る
        if(aMax <= 255){
            return;
        }
        //最大値が255になるようテーブルの各値を縮小する
        double aDivVal = static_cast<double>(aMax) / 255.0;
        for(aI = 0; aI < HC_ApTableSize; aI++){
            uint_32_t aNewVal = static_cast<uint_32_t>(
              static_cast<double>(mApTable[aI]) / aDivVal);
            if(aNewVal == 0 && mApTable[aI] != 0){
                aNewVal = 1;
            }
            mApTable[aI] = aNewVal;
        }
    }
```

リスト 11　HuffCompressクラス（符号化のメイン処理）

```
public:
    HuffCompress() {
        mHuffTreeRoot = NULL;
    }
    ~HuffCompress() {
        delete mHuffTreeRoot;
    }
    //圧縮を行う，成功すれば圧縮後のサイズを返す
    // 失敗すれば0を返す
    //iRBB=読み取りオブジェクト，iWBB=書き込みオブジェクト
    // iStartPos=読み取り開始位置(0～)
    // iProcSize=圧縮対象のサイズ
    uint_32_t compress(ReadByteBase& iRBB,WriteByteBase& iWBB,
                    uint_32_t iStartPos,int_32_t iProcSize) {
        //サイズが0以下なら戻る
        if(iProcSize <= 0){
            return 0;
        }
        //書き込み済みサイズをリセット
        iWBB.reset_wrote_size();
        //圧縮対象サイズを書き込む
        if(!iWBB.write_4byte(iProcSize)){
            return 0;
        }
        //出現頻度テーブルのクリア
        std::memset(mApTable,0,sizeof(mApTable));
        //読み取り開始位置をiStartPosに移動
        if(!iRBB.rewind_ptr(iStartPos)){
            return 0;
        }
        //出現頻度テーブルを作成する
        for(uint_32_t aCnt = 0; aCnt < static_cast<uint_32_t>(
              iProcSize); aCnt++){
            uint_8_t aDat;
            if(!iRBB.read_1byte(aDat)){
                return 0;
            }
            ++mApTable[aDat];
        }
        //出現頻度テーブルの値を0～255の範囲内におさめる
        shrink_ap_table();
        //出現頻度テーブルを書き込む
        for(int aI = 0; aI < HC_ApTableSize; aI++){
            if(!iWBB.write_1byte(static_cast<uint_8_t>(
                 mApTable[aI]))){
                return 0;
            }
        }
        //出現頻度テーブルから符号木を組み立てる
        if(!build_huff_tree()){
            return 0;
        }
        //符号木からハフマン・コード対応表を作成する
        build_huff_code_table();
        //読み取り開始位置を再びiStartPosに移動
        if(!iRBB.rewind_ptr(iStartPos)){
            return 0;
        }
        //ビット単位書き込みオブジェクトを用意する
        WriteBitClass aWBits(iWBB);
        //1バイトずつ読み込み，ハフマン符号化したものを
        // 書き込んでいく
        for(uint_32_t aCnt = 0; aCnt < static_cast<uint_32_t>(
              iProcSize); aCnt++){
            uint_8_t aDat;
            if(!iRBB.read_1byte(aDat)){
                return 0;
            }
            if(!aWBits.write_bits(mHuffCodeTable[aDat].len,
                 mHuffCodeTable[aDat].data)){
                return 0;
            }
        }
        //ビット単位書き込みオブジェクトをフラッシュする
        if(!aWBits.close_buffer()){
            return 0;
        }
        return iWBB.get_wrote_size();
    }
```

リスト 12　HuffCompress クラス（復号化のメイン処理）

```
    //展開を行う，成功すれば展開後のサイズを返す
    // 失敗すれば 0 を返す
    //iRBB=読み取りオブジェクト，iWBB=書き込みオブジェクト
    // iStartPos=読み取り開始位置(0～)
    uint_32_t uncompress(ReadByteBase& iRBB,
                    WriteByteBase& iWBB,uint_32_t iStartPos){
        //書き込み済みサイズをリセット
        iWBB.reset_wrote_size();
        //読み取り開始位置を iStartPos に移動
        if(!iRBB.rewind_ptr(iStartPos)){
            return 0;
        }
        //最初の 4 バイトを読む (復元サイズ)
        uint_32_t aOrigSize;
        if(!iRBB.read_4byte(aOrigSize)){
            return 0;
        }
        //次の 256 バイトを読む (出現頻度テーブル)
        for(int aI = 0; aI < HC_ApTableSize; aI++){
            uint_8_t aDat;
            if(!iRBB.read_1byte(aDat)){
                return 0;
            }
            mApTable[aI] = static_cast<uint_32_t>(aDat);
        }
        //出現頻度テーブルから符号木を組み立てる
        if(!build_huff_tree()){
            return 0;
        }
        //ルート・ノードがリーフである場合
        if(mHuffTreeRoot->is_leaf()){
            //復元サイズ分だけルート・ノードのシンボルを
            // 書き込む
            uint_8_t aDat = mHuffTreeRoot->symbol;
            for(uint_32_t aI = 0; aI < aOrigSize; aI++){
                if(!iWBB.write_1byte(aDat)){
                    return 0;
                }
            }
        }else{ //ルート・ノードがリーフでない場合
            //ビット単位の読み取りオブジェクトを生成する
            ReadBitClass aRBit(iRBB);
            uint_32_t aRecovSize = 0;
            //符号木をたどる追跡オブジェクト
            HuffTreeNodePtr aHuffTreeTracer = mHuffTreeRoot;
            //1ビットずつ読み取っていく
            while(aRecovSize < aOrigSize){
                uint_8_t aBit;
                if(!aRBit.read_1bit(aBit)){
                    return 0;
                }
                if(aBit){ //1である場合
                    //出現頻度の少ない側のノードへ
                    // 追跡オブジェクトを移動させる
                    aHuffTreeTracer = aHuffTreeTracer->
                                        littleFreq;
                }else{ //0である場合
                    //出現頻度の高い側のノードへ
                    // 追跡オブジェクトを移動させる
                    aHuffTreeTracer = aHuffTreeTracer->
                                        bigFreq;
                }
                //追跡オブジェクトはリーフにたどりついたかどうかを
                // 確認
                //リーフにたどりついた場合
                if(aHuffTreeTracer->is_leaf()){
                    //ノードのシンボルを書き込む
                    if(!iWBB.write_1byte(aHuffTreeTracer->
                                            symbol)){
                        return 0;
                    }
                    //1 バイトだけ復元した
                    ++aRecovSize;
                    //追跡オブジェクトをルート・ノードに戻す
                    aHuffTreeTracer = mHuffTreeRoot;
                }
            }
        }
        return iWBB.get_wrote_size();
    }
```

リスト 13　ハフマン圧縮クラス利用のサンプル

```
static void demoC() //HuffCompress::compressのテスト
{
    ReadByteFile aRBF;
    if(aRBF.open_file(TEST_DIR "sample.orig")){
        std::cout << "read open_file OK.¥n";
        WriteByteFile aWBF;
        if(aWBF.open_file(TEST_DIR "sample.huff")){
            std::cout << "write open_file OK.¥n";
            uint_32_t aFileSize;
            if(aRBF.get_file_size(aFileSize)){
                std::cout << "read file size = "
                          << aFileSize << std::endl;
                HuffCompress aHC;
                uint_32_t aCompSize = aHC.compress(aRBF,aWBF,
                                                0,aFileSize);
                std::cout << "compressed size = "
                          << aCompSize << std::endl;
            }else{
                std::cout << "read file size NG.¥n";
            }
            aWBF.close_file();
        }else{
            std::cout << "write open_file NG.¥n";
        }
        aRBF.close_file();
    }else{
        std::cout << "read open_file NG.¥n";
    }
}

static void demoU() //HuffCompress::uncompressのテスト
{
    ReadByteFile aRBF;
    if(aRBF.open_file(TEST_DIR "sample.huff")){
        std::cout << "read open_file OK.¥n";
        WriteByteFile aWBF;
        if(aWBF.open_file(TEST_DIR "sample.recov")){
            std::cout << "write open_file OK.¥n";
            HuffCompress aHC;
            uint_32_t aUncompSize = aHC.uncompress(aRBF,aWBF,
                                                          0);
            std::cout << "uncompressed size = "
                      << aUncompSize << std::endl;
            aWBF.close_file();
        }else{
            std::cout << "write open_file NG.¥n";
        }
        aRBF.close_file();
    }else{
        std::cout << "read open_file NG.¥n";
    }
}
```

はどうかと思われます．そこで記録するときは256バイト(1バイト×256)に圧縮してしまい，符号木の構築も256バイトに縮小した前提で行うようにします．そのために出現頻度の最大値を255以内に納める処理が別途必要になります(リスト10，p.179)．

圧縮処理で必要な情報として，圧縮前の情報のサイズあるいは圧縮後の情報のサイズがあります．これがないと復号化のときに，どこまで復号すべきかわからなくなって困るからです．符号化するときは「センチネル」[注5]を含める方法もありますが，ここでは圧縮前のサイズを記録します．つまり符号化したときの情報フォーマットは，

- 圧縮前のサイズ　　——4バイト
- 出現頻度テーブル　——256バイト
- 符号化ずみのデータ——必要なだけ

となります．

以上の準備ができれば符号化のメイン処理はさほど難しくありません．リスト11(p.179)のようになります．

ハフマン符号化の復号化処理部分

復号化処理は単純に考えると以下のような手順となります．

(1) シンボルの出現頻度テーブルを読み取り，符号木を組み立てる

(2) 符号化されたデータを読み取り，符号木を利用して元のシンボルに復元する

符号化に比べると復号化は案外，単純に実装できます(リスト12)．

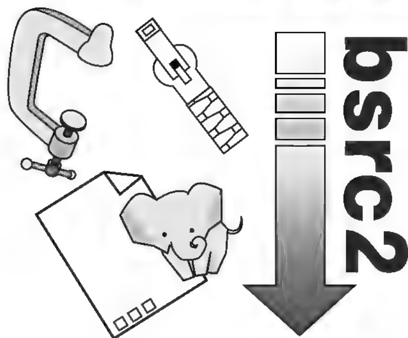
クラスどうしの組み合わせ

ここまで作成したクラスを組み合わせることでファイルを読み込んで圧縮したものを別のファイルに書き込んだり，すでに圧縮済みのファイルを読み込んで復元してから別のファイルに書き込む処理が簡単に実装できます．

リスト13が，その実例です．

みやさか・でんと　miyadent@anet.ne.jp

注5：sentinel，本来は「歩哨」という意味だが，プログラム上で処理を打ち切るときの目印となるデータや値の意味で使うことが多い．



マルチキー・クイック・ソートと0-1-2 codingにより高速化と高圧縮率を実現した

高性能圧縮ツールbsrcの改良bsrc2（後編）

三分割法と0-1-2 codingについて

広井 誠

ファイルを圧縮する場合，日本ではLHA，zip，gzipなどの圧縮ツールが一般に使用されている．圧縮ツールで用いられているデータ圧縮アルゴリズムであるブロック・ソート（Block Sorting）は圧縮性能が優れている方法で，これを用いた圧縮ツールbzip2[(10)]はLHA，zip，gzipよりも高い圧縮率になる．

筆者は本誌2003年12月号と2004年1月号で，ブロック・ソートとレンジ・コーダ（Range Coder）を用いた圧縮ツールbsrcを作成した．bsrcは，bzip2と同程度の圧縮率を達成したが，処理時間と圧縮率には改良の余地があった．

そこで，これらの改良を行ったプログラムbsrc2を作ったところ，処理時間はbsrcよりも大幅に短縮し，圧縮率はbzip2を上回り，zzip[(11)]に匹敵する性能を実現できた．本稿では，先月に引き続きこのbsrc2について解説を行う．（筆者）

先月号ではマルチキー・クイック・ソートと二段階ソート法による，速度面での改良を中心に解説しました．後編となる今回は，三分割法による速度のさらなる改善と，0-1-2 codingによる圧縮率の改善法について解説します．なお，今月号に掲載したプログラムは`http://www.cqpub.co.jp/interface/download/`からダウンロードできます．



三分割法の原理

先月解説した二段階ソート法は，ソートする個数を大幅に減らす画期的なアルゴリズムです．しかしながら，二段階ソート法にも弱点があります．一般に，バイナリ・データはあまり得意ではありません．とくに，同じ記号が多数続くデータ（「連長の多いデータ」という）や繰り返しの多いデータでは極端に遅くなります．

儘田真吾氏[(6)]によると，二段階ソート法を改良した「三分割法」を使うと，連長の多いデータでも高速にソートすることが可能になります．三分割法は記号S_iとS_{i+1}の大小関係を使って，サフィックスをType A，Type B，Type Cの3種類に分ける方法です．

Type A： $S_i > S_{i+1}$

Type B： $S_i = S_{i+1}$

Type C： $S_i < S_{i+1}$

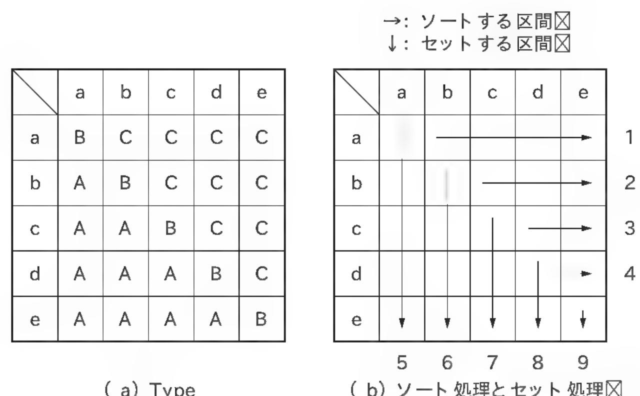
三分割法はS_iとS_{i+1}が等しい場合をType Bとし，$S_i<S_{i+1}$をType Cとします．そして，Type Cのサフィックスをソートするだけで，Type AとType Bのサフィックスの順序を決定することができます．Type Aのサフィックスの順序はType Cにより決定されますが，Type Bのサフィックスの順序はType Aから決定される部分とType Cから決定される部分の二つに分けることができます．このため，三分割法を「四分割法」と呼ぶ場合もあります．

それでは詳しく説明しましょう．図1に示すように，記号は｛a，b，c，d，e｝の5種類とします．三分割法の場合，区間のTypeは図1（a）のように分けることができます．三分割法はType Cの区間をソートし，その結果を使ってType AとType Bの記号列の順序を決定します．

まず最初に，二段階ソート法と同様にType Cの区間をソートします［図1（b）1，2，3，4］．次にType AとType Bのインデックスをセットします．このとき，Type Bの記号列の順序を決める処理が少し複雑になります．図2を見てください．

区間cを考えてみましょう．この場合，区間ca，cbがType A，区間ccがType B，区間cd，ceがType Cになります．Type Aの区間ca，cbは，二段階ソート法と同様に区間aとbのセット処理により，記号列の順序はすでに決定されています．Type Cの区間cd，ceをソートしたあと，Type Bの区間ccの記号列の順序を決定します．

区間ccの前半は区間ca，cbのセット処理によりインデックスがセットされます．ここでセットされるインデックスは図2のB1の部分だけです．B2の部分はB1の結果を使ってインデッ



	a	b	c	d	e
a	B	C	C	C	C
b	A	B	C	C	C
c	A	A	B	C	C
d	A	A	A	B	C
e	A	A	A	A	B

（a）Type

（b）ソート処理とセット処理

図1　三分割法

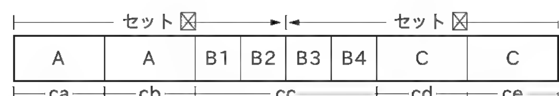


図2　Type Bの順序を決定する





クスをセットすることに注意してください．

これだけではなく，区間ccの後半は区間cd，ceからインデックスをセットする必要があります．このとき，図2に示すようにインデックスは後方からセットしていきます．つまり，区間cの先頭からだけではなく，後方からもType A，Type Bの記号列を探して，インデックスをセットするのです．

ここが三分割法を理解するポイントです．区間cdの先頭からType Bの記号列を探してインデックスをセットしようとすると，区間ccでインデックスをセットする位置が決定できないのです．区間cの後方から記号列を探していくことで，区間ccには後からインデックスをセットすることができます．つまり，区間ce，cdから区間ccのB4にインデックスがセットされ，区間ccのB4からB3にインデックスがセットされるわけです．これで，区間ccの記号列の順序を決定することができます．

このように，三分割法は記号xの区間xxをソートする必要がありません．したがって，連長の多いデータに三分割法を適用すれば，極めて高速にブロック・ソートすることができます．

● 三分割法の実装

三分割法のプログラムをリスト1に示します．

関数sort_two3()でType Cの区間をソートし，関数set_type_ab()でType A，Type Bの記号列を探してインデックスのセット処理を行います．このプログラムでは記号iのソートが完了したら，そのつどset_type_ab()を呼び出してType A，Type Bのインデックスをセットをしています．

関数set_type_ab()は，インデックスのセットを行う区間の先頭を配列startに，最後尾を配列endにセットします．今までのように，累積度数表count_sumを直接書き換えるとプログラムは正常に動作しません．三分割法では配列startとendの使い方がポイントになります．

次に，Type A，Type Bのインデックスをセットするため，記号symの区間を前から後へ探索します．ここで，symの区間にインデックスがセットされると，start[sym]の値が増えることに注意してください．これで探索の範囲が増えて，Type Bの区間にセットされたインデックスを使って，ほかの記号列のインデックスをセットすることができます．同様に，後から前へ探索するときはend[sym]の値が減少するので，探索の範囲が増えるわけです．これで，Type Bの区間にインデックスを正しくセットすることができます．

● 三分割法の改良

三分割法はType Cの区間をソートして，Type A，Type Bの区間にインデックスをセットします．儘田真吾氏(6)によると，三分割法はType Aの区間をソートして，Type B，Type Cの区間にインデックスをセットすることもできます(図3)．

最初にType Aの区間をソートします[図3(b)1，2，3，4]．次に，インデックスのセット処理を行いますが，今までとは逆に区間eから行うことに注意してください．最初に区間eを探索して区間ee，de，ce，be，aeにインデックスをセットします[図3(b)5]．次に，区間dを探索して，区間dd，cd，bd，adにインデックスをセットします[図3(b)6]．あとはこれを繰り返すだけです[図3(b)7，8，9]．

このように，二段階ソート法(三分割法)はType Aの区間をソートしても実現することができます．したがって，Type AとType Cの個数を求めて，個数が少ないTypeのほうをソー

リスト1　三分割法

```
/* Type A, B をセット */
void set_type_ab( int sym, int size )
{
  int i, start[CODE_SIZE], end[CODE_SIZE];
  /* 2 番目の記号が sym の区間を求める */
  for( i = sym; i < CODE_SIZE; i++ ){
    start[i] = count_sum[(i << 8) + sym];
    end[i]   = count_sum[(i << 8) + sym + 1] - 1;
  }
  /* 前から後へ */
  for( i = count_sum[sym << 8]; i < start[sym]; i++ ){
    Uchar *ptr = index_table[i];
    if( ptr == buffer ) ptr += size;
    if( *(ptr - 1) >= *ptr ){
      index_table[ start[*(ptr - 1)]++ ] = ptr - 1;
    }
  }
  /* 後から前へ */
  for( i = count_sum[(sym + 1) << 8] - 1; i > end[sym]; i-- ){
    Uchar *ptr = index_table[i];
    if( ptr == buffer ) ptr += size;
    if( *(ptr - 1) >= *ptr ){
      index_table[ end[*(ptr - 1)]-- ] = ptr - 1;
    }
  }
}

/* 三分割法 */
void sort_two3( int size )
{
  int i, j;
  for( i = 0; i < CODE_SIZE; i++ ){
    /* Type C をソート */
    for( j = i + 1; j < CODE_SIZE; j++ ){
      int low = count_sum[(i << 8) + j];
      int high = count_sum[(i << 8) + j + 1];
      if( high - low > 1 ){
        sort_count += high - low;
        mquick_sort( low, high - 1, 2, size );
      }
    }
    /* Type A, B をセット */
    set_type_ab( i, size );
  }
}
```

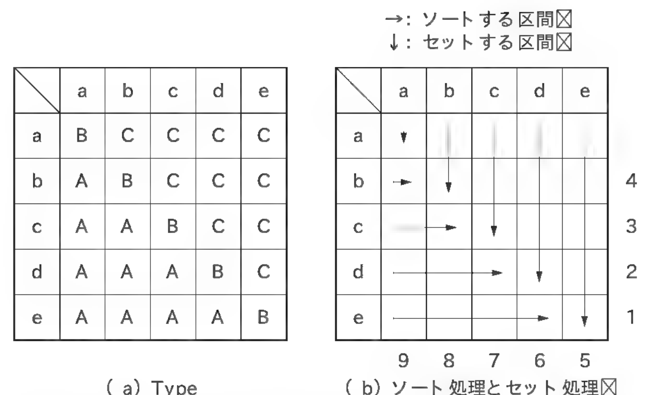


図3　三分割法でType Aをソートする場合

リスト 2　三分割法の改良

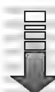
```
void sort_two31( int size )
{
  int i, a, c;
  /* Type A, C の個数をカウント */
  for( i = 0, a = 0, c = 0; i < CODE_SIZE; i++ ){
    a += count_sum[(i << 8) + i] - count_sum[i << 8];
    c += count_sum[(i + 1) << 8] - count_sum[(i << 8) + i + 1];
  }
  /* 少ない Type をソートする */
  if( a > c ){
    sort_two3( size );    /* Type C をソート */
  } else {
    sort_two3_a( size );  /* Type A をソート */
  }
}
```

トすれば，今までよりも処理時間を短縮できる場合があります．

プログラムをリスト 2に示します．

Type A の個数を変数 a に，Type C の個数を変数 c に求めます．記号を i とすると，Type A は区間（i，0）から（i，$i-1$）までになります．同様に，Type C は区間（i，$i+1$）から（i，0xff）までなので，累積度数表を使って簡単に計算することができます．

あとは，変数 a と c を比較して，c が少なければ関数 `sort_tow3()` を呼び出して Type C の区間をソートします．a が少なければ関数 `sort_two3_a()` を呼び出して，Type A の区間をソートします．`sort_two3_a()` は記号 0xff から Type A の区間をソートし，関数 `set_type_bc()` を呼び出して Type B，C のインデックスのセット処理を行います．



ratio-2 による三分割法の改良



三分割法にはもう一つ改良方法があります．儘田真吾氏[6]によると，三分割法に ratio-2 を適用することで，ソート処理をさらに高速化することができます．

三分割法に ratio-2 を適用する場合，二つの方法が考えられます．一つは最初の三分割を ratio-2 で行う方法です．もう一つは最初の三分割を ratio-1 で行い，そのあとで ratio-2 を適用する方法です．本稿では三分割法のプログラムを改造することで簡単に対応できる，後者の方法を採用することにします．

また，この方法でも ratio-2 に三分割法（$S_i > S_{i+2}$，$S_i = S_{i+2}$，$S_i < S_{i+2}$）を適用できると考えられますが，$S_i = S_{i+2}$ の部分にほかの Type と依存関係があるため，記号列の順序を決定できない場合があります．この部分を場合分けしてソートすることもできますが，プログラムがかなり複雑になると思われます．そのため，今回は ratio-2 は二分割にとどめます．ratio-2 の三分割は今後の課題にしたいと思います．

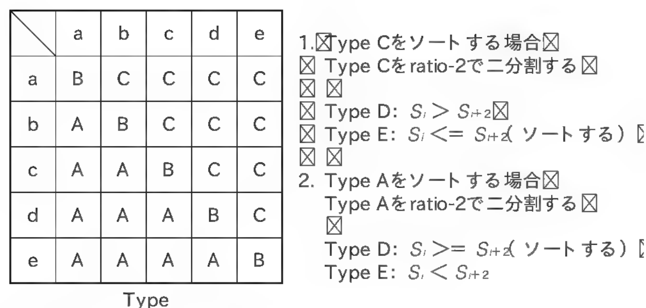
ratio-1 で三分割したあと ratio-2 を適用する場合，ソートするタイプで場合分けが異なります．

Type C をソートする場合，図 4に示すように Type C を ratio-2 で二分割します．この場合，記号列を $S_i > S_{i+2}$（Type D）と $S_i <= S_{i+2}$（Type E）に分けて，Type E をソートします．逆に，Type A をソートする場合，Type A を $S_i >= S_{i+2}$（Type D）と $S_i < S_{i+2}$（Type E）に分けて Type D をソートします．

この処理は図 5のように表すことができます．記号 b，c，d の部分だけを示します．

Type C に ratio-2 を適用すると，区間の前半が Type D で後半が Type E に分けられます．ソートするのは Type E だけで，Type D の記号列の順序はソート結果から決定できます（図 6）．

区間 cd の Type D は区間 cda と cdb があります．区間 cda のインデックスは区間 a からセットされ，区間 cdb のインデックスは区間 b からセットされます．Type C をソートする場合，小さな記号の区間からセット処理が行われるので，Type D の区間は前からインデックスがセットされていくことに注意してく



	a	b	c	d	e
a	B	C	C	C	C
b	A	B	C	C	C
c	A	A	B	C	C
d	A	A	A	B	C
e	A	A	A	A	B

図 4　三分割法に ratio-2 を適用（1）

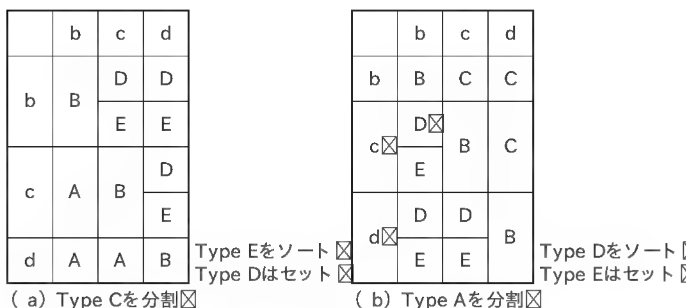


図 5　三分割法に ratio-2 を適用（2）

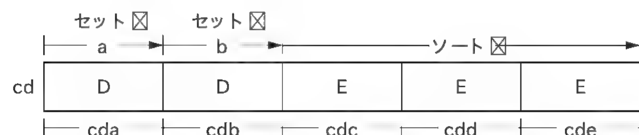


図 6　Type D の順序を決定する

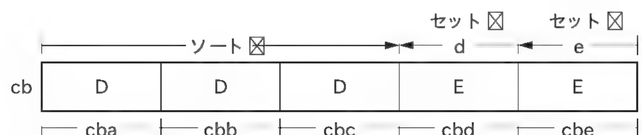


図 7　Type E の順序を決定する

リスト 3　三分割法(ratio-2)

```
void sort_two32( int size )
{
  int i, a, c, max;
  /* Type A, C の個数をカウント */
  for( i = 0, a = 0, c = 0, max = 0; i < CODE_SIZE; i++ ){
    int n = count_sum[(i << 8) + i] - count_sum[i << 8];
    if( max < n ) max = n;
    a += n;
    n = count_sum[(i + 1) << 8] - count_sum[(i << 8) + i + 1];
    if( max < n ) max = n;
    c += n;
  }
  /* work_table 領域のセット */
  work_table = get_buffer( sizeof( Uchar * ) * max );
  /* 少ないほうの Type をソートする */
  if( a > c ){
    sort_type_c2( size );
    set_type_ab2( size );
  } else {
    sort_type_a2( size );
    set_type_bc2( size );
  }
  free( work_table );
}
```

リスト 4　三分割法(ratio-2) Type Cをソート

```
/* ratio-2 用 */
int start2[CODE_SIZE * CODE_SIZE + 1];

void sort_type_c2( int size )
{
  int i, j;
  for( i = 0; i < CODE_SIZE; i++ ){
    /* Type C (Si < Si+1) を
       Type D (Si > Si+2) と Type E (Si <= Si+2) に
       分けてソートする */
    for( j = i + 1; j < CODE_SIZE; j++ ){
      int m, n, low = count_sum[(i << 8) + j];
      int high = count_sum[(i << 8) + j + 1];
      start2[(i << 8) + j] = low;       /* ratio-2 用 */
      for( m = n = low; n < high; n++ ){
        Uchar *ptr = index_table[n];
        if( *ptr > *(ptr + 2) ){
          index_table[n] = index_table[m];
          index_table[m++] = ptr;
        }
      }
      if( high - m > 1 ){
        sort_count += high - m;
        mquick_sort( m, high - 1, 2, size );
      }
    }
  }
}
```

ださい．

Type A に ratio-2を適用する場合も同じです．今度は区間の前半であるType Dをソートして，区間の後半であるType EはType Dのソート結果により記号列の順序が決定されます(図7)．

区間cbのType Eは区間cbdとcbeがあります．区間cbeのインデックスは区間eからセットされ，区間cbdのインデックスは区間dからセットされます．Type Aをソートする場合，大きな記号の区間からセット処理が行われるので，Type Eの区間は後からインデックスがセットされていくことに注意してください．

● 三分割法 ratio-2 の実装

ratio-2を適用した三分割法のプログラムをリスト 3に示します．

ratio-2用の作業領域にwork_tableを用意します．たとえば，Type Cをソートする場合，Type Dのインデックスは前から後へセットされます．セット処理は区間の前と後の両方向から行われるので，Type Dのセット方向と逆になることがあります．この場合，work_tableにインデックスを格納して，あとからType Dへインデックスをセットすることにします．work_tableの大きさはType AまたはType Cが連続する区間で，最大のものを変数maxに求めています．

次に，Type Cの個数が少ない場合は，sort_type_c()でratio-2を適用してType Eをソートし，set_type_ab()でインデックスのセット処理を行います．Type Aの個数が少ない場合は，sort_type_a()でソートして，set_type_bc()でインデックスをセットします．セット処理の前に記号列をType DとType Eに分けておかないとプログラムは正常に動作しません．

ソートを行う関数sort_type_c()をリスト 4に示します．

Type Cの記号列をソートする場合，Type Cの区間をType DとType Eに分けてからType Eの区間をソートします．Type Dは前から後へインデックスをセットしていくので，Type Dの先頭の位置を配列start2に格納しておきます．あとは記号*ptrと*(ptr+2)を比較して，記号列をType DとType Eに分離します．そして，Type Eの記号列をマルチキー・クイック・ソートでソートします．

Type Aをソートする場合は，記号の比較を*ptr<= *(ptr+2)とし，Type Eに$S_i = S_{i+2}$のデータを含めます．この場合，Type Dをソートして Type Eのインデックスをセットします．Type Eは後から前へインデックスをセットしていくので，start2にはType Eの最後尾の位置をセットします．

セット処理も難しくありません．リスト 5に示すように，リストは少々長いですが，今までのセット処理にratio-2用の処理が追加されているだけです．Type Eをソートして Type A，B，Dのインデックスをセットする場合，前から後へセットするときは簡単です．このとき，

```
*(ptr-2) < *ptr&& *(ptr-2) < *(ptr-1)
```

を満たせばratio-2のデータです．start2の位置へインデックスをセットします．

後から前へセットする場合は，ratio-2のインデックスをwork_tableにセットします．work_tableはスタックとして使用し，変数wpがスタック・ポインタになります．そのあとで，work_tableからデータを取り出して，start2の位置へインデックスをセットしていきます．

Type Dをソートして Type B，C，Eへインデックスをセットする場合は，前から後へインデックスをセットするときにwork_tableを使います．そして，ratio-2のデータの判定が，

```
*(ptr-2) < *ptr&& *(ptr-2) < *(ptr-1)
```

となることに注意してください．

リスト 5　三分割法(ratio-2) Type A，B をセット

```
void set_type_ab2( int size )
{
  int i, start[CODE_SIZE], end[CODE_SIZE];
  for( i = 0; i < CODE_SIZE; i++ ){
    int j, wp;
    for( j = i; j < CODE_SIZE; j++ ){
      start[j] = count_sum[(j << 8) + i];
      end[j]   = count_sum[(j << 8) + i + 1] - 1;
    }
    /* 前から後へ */
    for( j = count_sum[i << 8]; j < start[i]; j++ ){
      /* ratio-1 */
      Uchar *ptr = index_table[j];
      if( ptr == buffer ) ptr += size;
      if( *(ptr - 1) >= *ptr ){
        index_table[ start[ *(ptr - 1) ]++ ] = ptr - 1;
      }
      /* ratio-2 */
      ptr = index_table[j];
      if( ptr < buffer + 2 ) ptr += size;
      if( *(ptr - 2) > *ptr && *(ptr - 2) < *(ptr - 1) ){
        index_table[ start2[(*(ptr - 2) << 8)
                          + *(ptr - 1)]++ ] = ptr - 2;
      }
    }
    /* 後ろから前へ */
    for( wp = 0, j = count_sum[(i + 1) << 8] - 1; j > end[i];
         j-- ){
      Uchar *ptr = index_table[j];
      if( ptr == buffer ) ptr += size;
      if( *(ptr - 1) >= *ptr ){
        index_table[ end[ *(ptr - 1) ]-- ] = ptr - 1;
      }
      /* ratio-2 */
      ptr = index_table[j];
      if( ptr < buffer + 2 ) ptr += size;
      if( *(ptr - 2) > *ptr && *(ptr - 2) < *(ptr - 1) ){
        /* work_table に退避する */
        work_table[wp++] = ptr - 2;
      }
    }
    /* work_table からコピーする */
    while( wp > 0 ){
      Uchar *ptr = work_table[--wp];
      index_table[ start2[(*ptr << 8) + *(ptr + 1)]++ ] = ptr;
    }
  }
}
```

● 三分割法の評価結果

それでは，三分割法の評価結果を示しましょう．テスト・データはCanterbury Corpus(9)で配布されているThe Canterbury Corpusの中から alice29.txt，kennedy.xls，plrabn12.txt，ptt5 の四つのファイルとThe Large Corpusの bible.txt です．プログラムはBorland C++5.5.1 for Win32でコンパイルし，筆者のマシン(Windows 95，Pentium 166MHz)で実行しました．

結果を表1に示します．時間はファイルの入出力処理を含むプログラム全体の処理時間を示します．

ptt5 は0が連続しているデータで，二段階ソート法ratio-1(A)とratio-2(B)では極端に遅くなっています．このようなデータでも，三分割法(C)は高速にブロック・ソートすることができます．それ以外のデータでも，三分割法(C)とその改良版(D)はratio-1(A)よりもソートした個数が少なく，処理時間も速くなっています．三分割法の効果は十分に出ていると思います．

個数が少ないTypeをソートする三分割法の改良(D)は kennedy.xls で大きな効果がありました．個数が少ないタイプをソートする方法は，二段階ソート法を改善するのが簡単で効果的な方法だと思います．

三分割法ratio-2(E)は，どの方法よりもソートした個数が少なく，処理時間も速くなっています．三分割法にratio-2を適用した効果は十分に出ています．とくに，kennedy.xls ではratio-2の効果が大きく，ここまで速くなるとは筆者も予想していませんでした．

次に，バッファ・サイズを5Mバイトに増やしてみました．The Large CorpusとThe Canterbury Corpusの11ファイルをアーカイバtarでまとめたCanterburyの評価結果を表2に示します．

英文テキスト・ファイルの場合，二段階ソート法ratio-2(B)の効果が高いことがよくわかります．三分割法ratio-2(D)の場合，bible.txt と world192.txt でも(B)よりソートした個数が少なく，処理時間も速くなりました．三分割法にratio-2を適用した効果はとても高いことがわかります．

なお，これらの結果は筆者のコーディング，実行したマシン，コンパイラなどの環境に大きく依存しています．これらの環境の違いよって，処理時間はかなり左右されることに注意してください．

表1　三分割法の評価結果(バッファ 1Mバイト)

ファイル名	サイズ	(A)	(B)	(C)	(D)	(E)
alice29.txt	152,089	0.65 (74618)	0.64 (53507)	0.60 (67459)	0.60 (67459)	0.59 (48384)
kennedy.xls	1,029,744	4.86 (623977)	4.43 (467927)	3.99 (460331)	3.46 (404857)	2.94 (201688)
plrabn12.txt	481,861	2.09 (234016)	1.99 (162222)	2.03 (224466)	2.06 (224466)	1.99 (158938)
ptt5	513,216	201.1 (477580)	200.5 (465125)	1.23 (40306)	1.16 (34601)	1.16 (29788)
bible.txt	4,047,392	16.57 (1943800)	15.94 (1398313)	16.39 (1882603)	16.39 (1882603)	15.39 (1387678)

単位：秒．( )はソートしたデータ数，
バッファ 1Mバイト
(A)二段階ソート法(ratio-1)
(B)二段階ソート法(ratio-2)
(C)三分割法(ratio-1)
(D)三分割法(ratio-1)改良版
(E)三分割法(ratio-2)
注意：ソートはマルチキー・クイック・ソートを使用

表2　三分割法の評価結果(バッファ5Mバイト)

ファイル名	サイズ	(A)	(B)	(C)	(D)	(E)
bible.txt	4,047,392	20.36 (1943296)	18.72 (1398435)	20.14 (1882723)	20.14 (1882723)	18.3 (1387797)
Canterbury	2,821,120	195.82 (1775089)	192.08 (1394148)	9.69 (1082526)	9.53 (1067988)	8.57 (661763)
e.coli	4,638,690	30.69 (2861282)	26.48 (2192366)	23.24 (1642386)	23.24 (1642386)	20.48 (1437804)
world192.txt	2,473,400	17.6 (1326900)	13.72 (974228)	16.03 (1157123)	14.92 (1146161)	11.81 (778364)

単位：秒．( )はソートしたデータ数
(A)二段階ソート法(ratio-1)
(B)二段階ソート法(ratio-2)
(C)三分割法(ratio-1)
(D)三分割法(ratio-1)改良版
(E)三分割法(ratio-2)
注意：ソートはマルチキー・クイック・ソートを使用

このように，二段階ソート法を改良した三分割法を用いることで，ブロック・ソートを高速に行うことができます．

ところが，三分割法にも欠点があります．それは，繰り返しの多いデータでは極端に遅くなることです．このようなデータでも高速にソートできる方法にLarsson，Sadakane法があります．

そこで，bsrc2では，繰り返しの多いデータを検出したら，ブロック・ソートをLarsson，Sadakane法に切り替えるようにくふうしています．Larsson，Sadakane法はとても興味深いアルゴリズムですが，二段階ソート法によるブロック・ソートの高速化が本稿の趣旨なので説明は割愛します．URL(7)にはLarsson，Sadakane法を説明したスライドが掲載されています．また，岡野原大輔氏のWebサイト(13)や筆者のWebサイト(15)でもLarrson，Sdakane法を説明しています．興味のある方はこれらのWebサイトを参照してください．

0-1-2 coding

ブロック・ソートでデータを圧縮する場合，ブロック・ソートのあとMTF法やランレングスでデータを変換し，それをハフマン符号や算術符号(レンジ・コーダ)で符号化する方法が一般的です．ここで，ブロック・ソートに適した情報源モデルを作成すると，圧縮率が向上します．本稿では0-1-2 codingという方法を説明します．

0-1-2 codingは難しい方法ではありません．その名前が示すように，記号0，1，2を符号化する方法です．このとき，2以上の記号は2に変換して符号化するところがポイントです．つまり，記号を0，1，2の三つに分けて符号化するのです．

ブロック・ソートのあとデータをMTF法で変換すると，記号0と1の個数はとても多くなります．記号を0，1，2の3種類に限定すれば，高次の有限文脈モデルを適用することにより，圧縮率は極めて高くなります．あとは，残りの記号を適切な情報源モデルで符号化することにより，トータルで高い圧縮率を達成しようというのが0-1-2 codingの目的です．

0-1-2 codingの符号化のアルゴリズムをリスト6に示します．

このように，記号c1(0，1，2)をN次の有限文脈モデルで符号化し，記号cが2以上の場合は記号c2(c-2)を適切な情報源モデルで符号化します．復号も簡単で，まず記号c1を復号して値が2であれば，記号c2を復号して元の記号を求めるだけです．有限文脈モデルの次数ですが，試してみたところorder-3で十分のようです．

リスト6　0-1-2 codingの符号化(疑似コード)

```
encode()
{
  初期化処理
  do {
    c := ファイルから 1 記号読み込む
    c1 := c が 2 以上であれば 2 に変換する
    直前の N 記号から N 次モデルの出現頻度表 freq を求める
    記号 c1 をレンジ・コーダで符号化
    freq を更新する
    直前の記号を更新する
    if( c >= 2 ){
      /* 2 以上の記号の符号化 */
      c2 := c - 2
      c21 := 記号 c2 の first code を求める
      記号 c21 をレンジ・コーダで符号化
      if( c21 > 0 ){
        c22 := 記号 c2 の second code を求める
        記号 c22 をレンジ・コーダで符号化
      }
    }
  } while( 記号 != EOF );
  終了処理
}
```

表3
記号のグループ分け

GR	Num	Code
0	1	0
1	2	1，2
2	4	3 - 6
3	8	7 - 14
4	16	15 - 30
5	32	31 - 62
6	64	63 - 126
7	128	127 - 254(EOF)

記号c2をレンジ・コーダで符号化する場合，「無記憶情報源モデル」ではなく，適切な情報源モデルを使うことで圧縮率は向上します．そこで，記号をグループに分けて符号化する方法を使うことにします．この方法はbsrcでも使っています．表3を見てください．

Group 0は記号0だけですが，Group 1は1と2で，Group 2は3，4，5，6というように，グループに割り当てる記号の個数を増やしていきます．記号はEOFを含めて255(257-2)種類なので，これを0から7までのGroupに分けます．

次に，記号をGroup番号(first code)とGroup内の番号(second code)の二つに分けて符号化します．つまり，first

code(0-7)をレンジ・コーダで符号化し，次にsecond codeを符号化するのです．そして，second codeを符号化するとき，first codeによって記号の出現頻度表を切り替えます．このようなモデルをstructured modelと呼びます．

このように，0-1-2 codingは記号0，1，2に高次の有限文脈モデルを適用して圧縮率を向上させる方法です．このため，ランレングスとは相性があまりよくありません．とくにbsrcで用いたZero Length Encodingを適用した場合，0-1-2 codingではほとんど効果がありません．逆にいえば，ランレングスを適用しなくても，高い圧縮率を達成できる方法が0-1-2 codingなのです．

もっとも，データによってはランレングスが有効な場合もあります．たとえば，2以上の記号に対してランレングスを適用すると，圧縮率が向上する場合があります．bsrc2では，実際にランレングスを適用してみて，データが伸張しなければランレングスを用いるようにくふうしています．この方法はkennedy.xlsで大きな効果がありました．

● 出現頻度表の設定の調整

bsrc2は0-1-2 codingのほかに累積度数表の上限値と記号数の増分値を調整しています．

1) 出現頻度表の更新で，記号数の増分値を1から4に増やす
2) 累積度数表の上限値をモデルごとに設定する

適応型レンジ・コーダの場合，累積度数の上限値を少なめに，記号数の増分値を多めに設定すると，出現頻度表の更新が頻繁に行われるようになります．すると，最近出現している記号ほど出現確率が高くなり，データの変化にすばやく追随することができます．その結果，圧縮率が向上する場合があるのです．

この方法はブロック・ソートで驚くほど効果がありました．実際にstructured modelでfirst codeとsecond codeの上限値を調整すると，それだけでbsrcを上回る圧縮率になります．そして，0-1-2 codingとstructured modelを組み合わせることで，圧縮率はbzip2よりも高くなります．ここで，以下に説明する「混合法」という方法を適用すると圧縮率はさらに向上し，zzipに匹敵する圧縮率を達成することができます．

リスト7　有限文脈モデルの定義

```
/* 出現頻度表の定義 */
typedef struct {
  Ushort code_size, max_sum, inc_num;
  Ushort *count, *count_sum;
} Frequency;

/* 0-1-2 coding */
int c0000, c000, c00, c0;            /* 直前の記号を記憶 */
Frequency *freq_012_h[GR1][GR1][GR1][GR1]; /* order-4 */
Frequency *freq_012_l[GR1];          /* order-1 */
Frequency *freq_012_mix;             /* 混合用 */

/* structured model */
int f00, f0;                  /* first code : 直前の記号を記憶 */
Frequency *freq_first_h[GR2][GR2]; /* first code : order-2 */
Frequency *freq_first;             /* first code : order-0 */
Frequency *freq_first_mix;         /* 混合用 */
Frequency *freq_second[GR2];       /* second code */
```

● 混合法

混合法は複数のモデルを混合して得られる出現確率を使って記号を符号化する方法です．この方法には「文脈木重み付け法(Context-Tree Weighting：CTW)」があります．文脈木とは，情報源モデルを木構造で表したものです．簡単に説明すると，文脈木は節ごとに出現頻度表を持っていて，レベルNの節がN次の有限文脈モデルに対応します．したがって，文脈木の高さをHとすると，文脈木にはorder-0からorder-Hまでのモデルが存在することになります．

文脈木を使った符号化にはいくつか方法がありますが，その一つがCTWです．文脈木は優れたモデルですが，一般的なデータ(記号の種類が多いデータ)では，文脈木がとても大きくなってしまいます．このため，記号が2種類{0，1}の情報源に対して文脈木を適用する方法が一般的です．CTWも基本は2値の情報源に対してのものです．

CTWをプログラムするのは難しいので，bsrc2ではもっと単純な方法を使っています．高次の有限文脈モデルと低次の有限文脈モデルを用意します．そして，高次のモデルと低次のモデルから出現頻度表を選択し，二つの出現頻度表を単純に加算することで，記号の出現確率を求めることにします．とても単純な方法ですが，0-1-2 codingとsturctured modelのfirstcodeに適用してみたところ，とても高い効果が得られました．

● 0-1-2 codingと混合法の実装

具体的には，0-1-2 codingのorder-4とorder-1を混合し，first codeのorder-2とorder-0を混合します．プログラムをリスト7に示します．

出現頻度表は構造体Frequencyで定義します．0-1-2 codingの場合，freq_012_hが高次の有限文脈モデル(order-4)で，freq_012_lが低次の有限文脈モデル(order-1)を表します．そして，二つの出現頻度表を混合するためfreq_012_mixを用意します．

structured modelのfirst codeも同様に，freq_first_hがorder-2の有限文脈モデルで，freq_firstがorder-0の有限文脈モデルになります．そして，混合用にfreq_first_mixを用意します．

出現頻度表の混合は簡単です．プログラムをリスト8に示します．

関数mix_012_frequency()は，0-1-2 codingのorder-4とorder-1から出現頻度表を選び，それらの出現頻度表の値を加算してfreq_012_mixにセットします．直前の記号列はグローバル変数c0000，c000，c00，c0に格納されているので，出現頻度表は簡単に選択できます．同様に，first code用の関数mix_first_frequency()もorder-2から出現頻度表を選択し，order-0の出現頻度表と加算して，その値をfreq_first_mixにセットするだけです．

0-1-2 codingとfirst codeの符号化は，混合した出現頻度表を使って行います．プログラムをリスト9に示します．

リスト 8　出現頻度表の混合

```
/* 0-1-2 coding : 混合用 */
Frequency *mix_012_frequency( void )
{
  int i;
  Frequency *freq1 = freq_012_h[c0000][c000][c00][c0];
  Frequency *freq2 = freq_012_l[c0];
  Ushort *count1 = freq1->count, *count2 = freq2->count;
  Ushort *count  = freq_012_mix->count;
  Ushort *count_sum = freq_012_mix->count_sum;
  count_sum[0] = 0;
  for( i = 0; i < GR1; i++ ){
    count[i] = count1[i] + count2[i];
    count_sum[i + 1] = count_sum[i] + count[i];
  }
  return freq_012_mix;
}

/* first code : 混合用 */
Frequency *mix_first_frequency( void )
{
  int i;
  Frequency *freq1 = freq_first_h[f00][f0];
  Ushort *count1 = freq1->count, *count2 = freq_first->count;
  Ushort *count = freq_first_mix->count;
  Ushort *count_sum = freq_first_mix->count_sum;
  count_sum[0] = 0;
  for( i = 0; i < GR2; i++ ){
    count[i] = count1[i] + count2[i];
    count_sum[i + 1] = count_sum[i] + count[i];
  }
  return freq_first_mix;
}
```

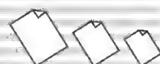
最初は0-1-2 codingです．記号cが2以上の場合，値を2に変換してc1にセットします．そして，mix_012_frequency()でorder-4とorder-1を混合した出現頻度表を作成し，その出現頻度表freqを使って適応型レンジ・コーダで符号化します．RC_ENCODEは適応型レンジ・コーダで符号化を行うマクロです．出現頻度表の更新は関数update_mix_012_frequency()で行います．

次に，cが2以上の場合はstructured modelで符号化を行います．まず最初にfirst codeを符号化します．mix_first_frequency()でorder-2とorder-0を混合した出現頻度表を作成し，その出現頻度表を使ってRC_ENCODEで符号化します．そして，first code(c1)が0より大きければsecond codeを符号化します．

それから，出現頻度表を更新するとき，二つのモデルで記号数の増分値を変えると，圧縮率が向上する場合があります．0-1-2 codingではorder-4を2，order-1を14とし，first codeではorder-2を4，order-0を12としたところ，良好な結果が得られました．



bsrc2の評価



それではbsrc2の評価結果を示します．圧縮率を比較するために，bsrc，bzip2[10]，szip[12]，zzip[11]の評価結果も示します．これらの圧縮ツールはすべてブロック・ソートを使っています．テスト・データはCanterbury Corpus[9]で配布されているThe Canterbury CorpusとThe Large Corpusを使いました．The Canterbury Corpusの評価結果を表4に示します．

リスト 9　0-1-2 coding, structured model，混合法による符号化

```
int range_encode( Uchar *out, Uchar *in, int size )
{
  int i, rp, wp, c;
  Uint range = ULONG_MAX, low = 0;
  /* 初期化 */
  init_frequency();
  rp = wp = 0;
  do {
    Frequency *freq;
    int c1, c2;
    if( rp < size ){
      c = in[rp++];
    } else {
      c = END;
    }
    /* 0-1-2 coding : 混合法 */
    c1 = (c < 2 ? c : 2);
    freq = mix_012_frequency();
    RC_ENCODE( freq, c1, range, low, wp, out );
    update_mix_012_frequency( c1 );
    /* structured model */
    if( c >= 2 ){
      /* 記号をグループに変換 */
      c1 = group_table[c - 2];
      c2 = (c - 2) - group_low[c1];
      /* first code の符号化 : 混合法 */
      freq = mix_first_frequency();
      RC_ENCODE( freq, c1, range, low, wp, out );
      update_mix_first_frequency( c1 );
      if( c1 > 0 ){
        /* second code の符号化 */
        freq = freq_second[c1];
        RC_ENCODE( freq, c2, range, low, wp, out );
        update_frequency( freq, c2 );
      }
    }
  } while( c != END );
  /* 最後の出力 */
  for( i = 0; i < 4; i++ ){
    out[wp++] = low >> 24;
    low <<= 8;
  }
  return wp;
}
```

bsrc2はbsrc，bzip2，szipよりも高い圧縮率になりました．zzipはkennedy.xlsに対して特別な操作を行っているため，極端に圧縮率が高くなっています．ほかのファイルではbsrc2のほうが高い圧縮率になりましたが，zzipで-mx(MaXimum compression)を指定すると，zzipの圧縮率のほうが高くなります．

次はThe Large Corpusの評価結果を示します．バッファの大きさを1Mバイトに設定した結果を表5に，4Mバイトに設定した結果を表6に示します．CanterburyはThe Canterbury Corpusの11のファイルをアーカイバtarでまとめたものです．

バッファの大きさがどちらの場合でも，bsrc2の圧縮率はzzip -mxを上回り，とても高い圧縮率を達成しています．bsrc2では0-1-2 codingと混合法による改良を行いましたが，その効果は十二分に出ています．どちらも簡単な方法ですが，ここまで圧縮率が向上するとは予想していなかったので，筆者もたいへん驚いています．

次は圧縮の処理時間を示します．バッファの大きさを1Mバイトに設定した結果を表7に，4Mバイトに設定した結果を

表4　The Canterbury Corpus の評価結果

ファイル名	サイズ	bzip2	bsrc	bsrc2	szip	zzip	zzip -mx
alice29.txt	152,089	43,202 (28.4)	43,230 (28.4)	41,420 (27.2)	42,835 (28.2)	41,962 (27.6)	41,242 (27.1)
asyoulik.txt	125,179	39,569 (31.6)	39,919 (31.9)	38,308 (30.6)	39,431 (31.5)	38,747 (31.0)	38,464 (30.7)
cp.html	24,603	7,624 (31.0)	7,625 (31.0)	7,404 (30.1)	7,541 (30.7)	7,509 (30.5)	7,393 (30.0)
fields.c	11,150	3,039 (27.3)	3,023 (27.1)	2,918 (26.2)	3,086 (27.7)	2,971 (26.6)	2,953 (26.5)
grammar.lsp	3,721	1,283 (34.5)	1,244 (33.4)	1,192 (32.0)	1,232 (33.1)	1,243 (33.4)	1,237 (33.2)
kennedy.xls	1,029,744	130,280 (12.7)	99,949 (9.7)	88,329 (8.6)	107,591 (10.4)	24,778 (2.4)	24,778 (2.4)
lcet10.txt	426,754	107,706 (25.2)	108,013 (25.3)	103,052 (24.1)	106,884 (25.0)	104,193 (24.4)	102,392 (24.0)
plrabn12.txt	481,861	145,577 (30.2)	146,601 (30.4)	139,826 (29.0)	143,631 (29.8)	141,418 (29.3)	139,632 (29.0)
ptt5	513,216	49,759 (9.7)	48,794 (9.5)	47,293 (9.2)	52,858 (10.3)	48,329 (9.4)	48,329 (9.4)
sum	38,240	12,909 (33.8)	12,599 (32.9)	12,123 (31.7)	12,851 (33.6)	12,297 (32.2)	12,297 (32.2)
xargs.1	4,227	1,762 (41.7)	1,711 (40.5)	1,657 (39.2)	1,739 (41.1)	1,710 (40.5)	1,696 (40.1)
合計	2,810,784	542,710 (19.3)	512,708 (18.2)	483,522 (17.2)	519,679 (18.5)	425,157 (15.1)	420,413 (15.0)

単位：バイト．( )は圧縮率[%]
圧縮ツールで使用したオプション：
```
bzip2 -b9,
bsrc -e,
bsrc2 -e,
szip -b17,
zzip a -1m,
zzip a -1m -mx
```

表5　The Large Corpus の評価結果（バッファ・サイズ 1M バイト）

ファイル名	サイズ	bsrc	bsrc2	szip	zzip	zzip -mx
bible.txt	4,047,392	841,671 (20.8)	802,465 (19.8)	840,582 (20.8)	811,239 (20.0)	804,543 (19.9)
Canterbury	2,821,120	523,347 (18.6)	496,897 (17.6)	530,564 (18.8)	539,553 (19.1)	539,553 (19.1)
e.coli	4,638,690	1,233,563 (26.6)	1,156,896 (24.9)	1,174,473 (25.3)	1,167,753 (25.2)	1,167,753 (25.2)
world192.txt	2,473,400	482,329 (19.5)	464,671 (18.8)	508,380 (20.6)	469,101 (19.0)	466,715 (18.9)
合計	13,980,602	3,080,910 (22.0)	2,953,262 (21.1)	3,053,999 (21.9)	2,987,646 (21.4)	2,978,564 (21.3)

単位：バイト．( )は圧縮率[%]
圧縮ツールで使用したオプション：
```
bsrc -e,
bsrc2 -e,
szip -b11,
zzip a -1m,
zzip a -1m -mx
```

表6　The Large Corpus の評価結果（バッファ・サイズ 4M バイト）

ファイル名	サイズ	bsrc	bsrc2	szip	zzip	zzip -mx
bible.txt	4,047,392	798,772 (19.7)	759,499 (18.8)	792,221 (19.6)	767,895 (19.0)	763,577 (18.9)
Canterbury	2,821,120	539,905 (19.1)	499,871 (17.7)	530,346 (18.8)	543,760 (19.3)	554,078 (19.6)
e.coli	4,638,690	1,226,161 (26.4)	1,153,779 (24.9)	1,173,895 (25.3)	1,164,598 (25.1)	1,164,598 (25.1)
world192.txt	2,473,400	429,774 (17.4)	412,714 (16.7)	467,664 (18.9)	417,108 (16.9)	415,422 (16.8)
合計	13,980,602	2,994,612 (21.4)	2,825,863 (20.2)	2,964,126 (21.2)	2,893,361 (20.7)	2,897,675 (20.7)

単位：バイト．( )は圧縮率[%]
圧縮ツールで使用したオプション：
```
bsrc -e -b8,
bsrc2 -e -b8,
szip -b41,
zzip a -4m,
zzip a -4m -mx
```

表7　圧縮の処理時間（バッファ・サイズ1Mバイト）

ファイル名	サイズ	bsrc	bsrc2	szip	zzip	bzip2
bible.txt	4,047,392	32.26	29.70	17.67	31.83	22.97
Canterbury	2,821,120	18.94	17.84	10.72	18.10	12.28
e.coli	4,638,690	50.97	35.78	18.60	41.96	25.47
world192.txt	2,473,400	21.02	19.77	10.68	21.30	14.80

単位：秒
圧縮ツールで使用したオプション：
bsrc -e，bsrc2 -e，szip -b11，zzip a -1m

表8　圧縮の処理時間（バッファ・サイズ4Mバイト）

ファイル名	サイズ	bsrc	bsrc2	szip	zzip
bible.txt	4,047,392	38.04	32.27	18.87	35.52
Canterbury	2,821,120	21.34	18.74	10.76	19.34
e.coli	4,638,690	59.08	38.68	19.64	48.15
world192.txt	2,473,400	24.53	20.03	11.23	25.03

単位：秒
圧縮ツールで使用したオプション：
bsrc -e -b8，bsrc2 -e -b8，szip -b41，zzip a -4m

表8に示します．bzip2のバッファは最大で900Kバイトまでしか設定できないので，今回のテストはbzip2が少々有利であることに注意してください．

szipのブロック・ソートは「限定ソート法」という特別な方法を使っています．これは，完全なブロック・ソートを行わずに，比較する記号列の長さを限定することで，高速にブロック・ソートを行う方法です．このため，szipの処理時間が一番速くなっています．ただし，限定ソート法はブロック・ソートの復号処理が複雑になるという欠点があります．

bsrc2はbsrcとzzipよりも高速ですが，bzip2にはかないません．これは適応型レンジ・コーダの符号化に時間がかかっているからです．bsrc2は0-1-2 codingと混合法を用いているため，適応型レンジ・コーダの処理時間はbsrcのそれよりも約3倍かかります．圧縮性能が高くなった分だけ符号化に時間がかかるようになりました．それでも，bsrc2はzzipよりも高速なので，三分割法によるブロック・ソートの改良は，十分に効果を発揮していると思います．

おわりに

bsrcを改良した圧縮ツールbsrc2を作成しました．ブロック・ソートの高速化では，マルチキー・クイック・ソートとsuffix arrayの構築アルゴリズムである二段階ソート法（三分割法）を用いることで，圧縮の処理時間をbsrcよりも短縮することができました．また，圧縮率の改良では，0-1-2 codingと混合法を用いることにより，bsrcとbzip2を上回りzzipに匹敵する圧縮性能を達成することができました．

ところで，bsrc2で試した方法以外にも，ブロック・ソートの圧縮率を向上させる方法があります．たとえば，MTF法に代わるアルゴリズムがいくつか考案されています．IF（Inversion Frequencies），AWFC（Advanced Weighted Frequency count），DC（Distance Coding）などが有名で，これらのアルゴリズムを用いることでbsrc2やzzipを上回る圧縮率を達成することが可能です．興味のある方は，これらのアルゴリズムを調べてみてください．yuu氏のWebサイト(14)では，IF，AWFC，DCなどブロック・ソート関連のプログラムが公開されているので参考になると思います．

最後に，本稿がデータ圧縮アルゴリズムに関心のある読者の参考になれば幸いです．

● 権利・免責事項など

本稿で作成したプログラムbsrc2はフリー・ソフトウェアとします．ご自由にお使いください．ただし，これらのプログラムは無保証であり，使用したことにより生じた損害について，筆者は一切の責任を負いません．また，これらのプログラムを販売することで利益を得るといった商行為は禁止します．

bsrc2はデータ圧縮アルゴリズム評価用のサンプル・プログラムであり，ファイルの安全性はまったく考慮しておりません．実用的な圧縮ツールとして使用しないようにご注意願います．

参考文献，URL
(1) 植松友彦；文書データ圧縮アルゴリズム入門，CQ出版社，1994年
(2) 奥村晴彦；C言語による最新アルゴリズム事典，技術評論社，1991年
(3) 奥村晴彦；データ圧縮の基礎から応用まで，C MAGAZINE 2002年7月号，ソフトバンク
(4) Jon Bentley，Robert Sedgewick，*Fast Algorithms for Sorting and Searching Strings*，http://www.cs.princeton.edu/~rs/strings/
(5) 伊東秀夫；Suffix Arrayの効率的な構築法，http://www.ricoh.co.jp/rdc/techreport/No27/
(6) 儘田真吾；suffix arrayの構築──二段階ソート法とその改良，http://www.isl.cs.gunma-u.ac.jp/~shingo/algo.html
(7) 定兼邦彦；大規模テキスト索引（suffix array）の構築法とその情報検索への応用 suffix array構築アルゴリズムと実装，http://www.gi.k.u-tokyo.ac.jp/ssr-homepage/1999/workshop1/
(8) 山本博資；ユニバーサルデータ圧縮アルゴリズムの変遷─基礎から最新手法まで─，http://hirosuke.sr3.t.u-tokyo.ac.jp/files/survey.html
(9) Canterbury Corpus，http://corpus.canterbury.ac.nz/
(10) The bzip2 and libbzip2 home page，http://sources.redhat.com/bzip2/
(11) Zzip，http://debin.org/zzip/
(12) szip homepage，http://www.compressconsult.com/szip/
(13) DO++，http://member.nifty.ne.jp/DO/
(14) white page，http://homepage3.nifty.com/wpage/
(15) M.Hiroi's Home Page，http://www.geocities.co.jp/SiliconValley-Oakland/1680/

ひろい・まこと

オープンソースのITRON仕様OS

TOPPERSで学ぶRTOS技術

第9回 JSPカーネル移植のための基礎知識

邑中 雅樹

2004年8月号までの本連載では，シミュレーション環境を用いて，おもにアプリケーションを書くための解説が続きました．JSPカーネルは，μITRON4.0仕様スタンダード・プロファイルに準拠しているので，8月号までの内容は，TOPPERSカーネル以外のμITRON4.0仕様OSでも利用できます．

今回からの数回は，アプリケーション・プログラミングの世界から少し離れて，カーネル内部の話題に移ります．TOPPERSカーネルに固有の話が多くなるかもしれませんが，どのリアルタイム・カーネルもおおむね似たような構造をもっているので，ほかのリアルタイム・カーネルを扱うときにも参考になるのではないかと思います．

カーネル内部を触る動機としては，

- サービス・コールを拡張したい
- 新しいボードやCPUに対応したい

といった理由が考えられますが，今回は，配布パッケージに含まれていない新しいターゲット・ボードへの移植方法を紹介します．

そのために，まずは移植に必要な道具について解説します．アプリケーション開発とカーネル移植では必要な知識が少し異なり，オープン・ソース固有の事情やコンパイラやデバッガに関する，知られているようで知られていない留意点もいくつかあるので，少し長めに解説しました．

次に，今回移植する環境の紹介を行い，最低限の環境構築を試みます．

8月号までの内容とは異なり，今回以降の連載はJSPカーネルに強く依存した話になります．可能な限りソース・コードを誌面に引用するようにしますが，お手元のPCにJSPカーネルのソース・コードを展開しておくことを勧めます．

JSPカーネルの構成

JSPカーネルは，当初から移植性を重視しているため，C言語で記述されたハードウェアに依存しない部分と，アセンブラが必要だったり周辺デバイスへのアクセスが必要になるターゲット依存部分とが，ソース・コードのレベルで，明確に分離されています．さらに，ターゲット依存部は，CPU依存部とボード依存部に明確に分かれます．また，Release1.4からは，GCC以外のビルド環境への対応が強化され，一部が分離されました．

これらをまとめると，JSPカーネルが想定するターゲットのモデルは図1のようになっていると考えられます．JSPカーネルの移植作業とは，MPU，ボード，ツールに関して依存している部分を，使っているビルド・ツールとターゲットにあわせこむ作業といえます．

では，具体的にどのように分離されているのかをざっと見てみましょう．

図2は，JSPカーネルのディレクトリ構成を抜粋したものです．ターゲット依存部は，`config`ディレクトリにまとめられています．また，ターゲット依存部は，CPU依存部とボード依存部に分かれています．たとえば，`config/h8/akih8_3048f`というディレクトリがありますが，この場合，`config/h8`ディレクトリの中にGCC用のH8に依存する部分が，`akih8_3048f`ディレクトリの中に秋月電子通商製H8/3048ボードに依存する部分が，それぞれ格納されています．

CPU依存部やボード依存部に含まれるコードの分量は，ターゲットによって大きく異なります．SHのように周辺ペリフェラルが内蔵され，ある程度標準化されている場合にはCPU依存部が大きく，ボード依存部はメモリ・マップの定義程度になります．一方，ARMのように，チップごとに構成が大きく異なる場合や，IA32のように周辺ペリフェラルが外付けの場合には，ボード依存部の比重が大きくなります．

JSPカーネルの移植の難易度は，標準配布のJSPカーネルに付属しているツール依存部・ターゲット依存部（CPU依存部/ボード依存部）と，移植対象のビルド・ツール/ターゲット・

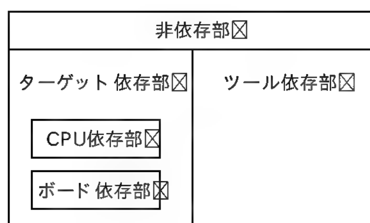


図1 JSPカーネルが想定するターゲットのモデル

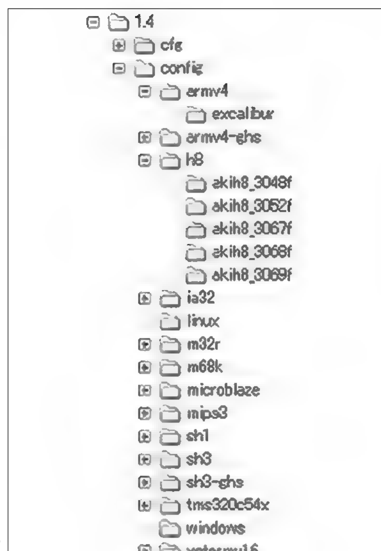


図2
configディレクトリの構成

ボードとがどれだけ似ているかという点にかかってきます．

これらのディレクトリ構成には，いくつかの例外もあります．Release1.4の時点では，ツールに関してCPUとボードの関係ほどには明確に分離されていません．ツールの違いは，CPUの違いとして整理されています．CPU依存部にはアセンブラ・コードが多く含まれていますが，アセンブラの文法はツールによって大きく異なるのが一般的であるため，このような区分になっているようです．JSP1.4を例に取ると，`armv4`（GCC版）と`armv4-ghs`（GreenHills社製開発環境用）というように，別のCPUとして扱うようになっています．このあたりは将来のリリースで修正が入る可能性があります．また，LinuxやWindowsのように，ほかのOS上でエミュレーション動作をする場合も，CPUとボードの区別は曖昧になります．

JSPカーネルを移植するために必要な道具

まずは一般論として，必要な道具についてまとめてみます．

● 動作するターゲット・ボードを用意する

当然といえば当然なのですが，「動作する」という点が，非常に重要です．

ターゲットのハードウェア的なバグを発見する可能性は，カーネルの移植中のほうがアプリケーション開発よりも高くなります．アプリケーション開発を主業務にしている方は，いつもより余計に「渡されたボードはバグだらけかもしれない」と疑ってかかってみてください．

● コンパイラを選ぶ

これも当然なのですが，TOPPERSのusers-MLを見ていると，コンパイラでつまづく人が後を絶たないようです．

▶ まずは推奨コンパイラで移植する

`doc`ディレクトリには，推奨コンパイラに関する情報があるので必ず一読しましょう．ドキュメントがGCC以外を推奨している場合は別ですが，初めての移植の時には，GCCを選択することを強くお勧めします．

GCCは移植性が高い反面，CPU固有の機能を使いきれないという弱点があります．そのため，チップ・メーカが提供するコンパイラが必要になることがあります．しかし，一気にコンパイラの変更をともなう移植作業に入るのは，かなり敷居の高い作業です．

JSPは可能な限り移植性を確保しているものの，ターゲット依存部によっては，特定のコンパイラに強く依存したコードを含んでいます．こういった部分で移植にともなうバグが発生すると，原因の切り分けが難しくなります．TOPPERSカーネルのサポートを提供している企業と契約を結んでいる場合は別ですが，一般的に原因の切り分けに悩んだときには，メーリング・リストなどに頼ることになります．そのターゲットでは非推奨のコンパイラで問題が発生した場合，持っている人が少ないので，ヒントや回答を得られる可能性が低くなります．

▶ GCCのバージョンに注意

GCCを使う場合には，バージョンにも注意しましょう．GCCはマイナ・バージョンが違えば別のコンパイラである，というくらいの覚悟で臨んでもよいくらいです．

よくある例として，GCC 2.95x系用のコードがGCC 3.x系で動かないというものがあります．また，GCC 3.4系では，インライン・アセンブラ周辺の挙動が変わり，JSP 1.4カーネルがビルドできなくなったという情報もあります．注意が必要です．

▶ バイナリ配布を有効活用しよう

GCCは無償でダウンロードできますし，クロス・コンパイラ構築を解説するWebページも増えてきています．しかし，それでも自分でビルドしないで済む方法がないかどうか検討してみることを勧めます．

一般に，オープンソース・ツールのビルドには，組み込み開発者にはなじみの薄い雑多な知識が要求されます．そのため，LinuxやFreeBSDなどオープンソースのOSでも，バイナリ・ディストリビューションが一般的になりました．しかし

コラム 1 源流になったコード

TOPPERSカーネルは，多くの開発メンバによってメンテナンスされています．メンバも日々精進していますが，残念ながらバグを作り込んでしまうことは避けられません．機種依存部のコードを読んでいてバグかどうか悩んだときに，開発メンバはm68k/dve68kターゲットの挙動を判断材料としています．このターゲットは，名古屋大学の高田広章教授がJSPカーネルを作成するにあたって最初に書いたものであり，ほかのすべてのターゲットは，ここから派生しています．

GCCは，数あるオープンソース・ツールの中でも，正確にビルドするためには知識が数多く必要です．binutilsやnewlibなどほかのツールが必要で，それらの間での相性問題が出ることがあります．とくにWindows上ではCygwinのバージョンなども絡みます．

GCCは数多くのCPUに対応した代償として，特定のターゲットで動作が怪しくなるケースも散見されます．GCCのバージョンによっては，特定のホストとターゲットではビルドさえできない場合もあります．

このように，数多くの困難があるため，つねにアンテナを張って最新の情報を収集できる自信がある人以外には，ソースからビルドすることは勧められません．幸い，いくつかのボード・メーカではCygwinなどの環境でビルドしたGCCを製品に添付しています．ボード選択の際には，GCCが添付しているかどうかを選定の基準にするのも良いと思います．また，一部のソフト・ハウスは，組み込み技術者向けにGCCのバイナリをパッケージ化して販売，もしくは無償配布しています．こういったバイナリ・ディストリビューションを利用すると，移植作業を迅速に開始できます．

● デバッガは必須のアイテム

JSPカーネルの採用が検討されるような規模の開発案件では，デバッガなしでの開発を行っているケースもあると思うのですが，カーネルの移植においてデバッガは必須だと考えてください．

▶ICEは必須ではないが…

ICEがあれば工数が格段に短縮されますが，必須ではありません．実際，JSPカーネルの最初の版は，ほぼICEなしで開発されました．とはいえ，個人的感想としては，ICEがあると移植作業の効率がたいへんよくなります．JTAGが普及してくれたおかげでICEの値段も下がっており，筆者のような零細企業でもなんとか買えるようになってきています．

▶「TOPPERS対応」?

いくつかのツール・メーカが提供するデバッガは，TOPPERSカーネル対応をうたっています．これらは，カーネル・オブジェクトの内容表示やシステム・コール呼び出し情報の収集を行う機能をもって対応としているようです．

しかし，カーネル自身の移植に限っていえば，デバッガがTOPPERSに対応しているかどうかはあまり重要ではありません．というのも，カーネル移植時に注目するコードの大半が，カーネル・オブジェクトの初期化が行われる前か，ほとんどのサービス・コールが使えない非タスク・コンテキストだからです．TOPPERSカーネル対応のいかんよりも，移植対象ボードやCPUでの実績があることのほうが重要です．

もちろん，カーネル移植後のアプリケーション開発では心強い機能となります．先を見越して対応製品を用意するというのであれば，TOPPERS対応を表明している製品を勧めます．

▶DWARF2の罠(?)に注意

コンパイラとしてGCCをお勧めしたことと関連するのですが，デバッガのフロントエンドにも注意が必要です．GCCが生成するデバッグ情報はDWARF2です．ほかのデバッガでもDWARF2に対応しているはずなのですが，GCCは(仕様には準拠しているそうですが)固有の拡張情報を含むようで，ICEメーカが提供する独自デバッガでは拡張情報をうまく扱えないケースがあります．デバッガを選定する際には，GCCのどのバージョンに対応しているのかを確認する必要があります．

筆者の個人的な見解なのですが，GDB(GNU debugger)をベースとするものを勧めます．GCCとGDBの間でも，まれにバージョンの不一致でデバッグ情報を正しく読めないケースがあるようですが，それでも，問題が出る可能性は低いと思われ

コラム2　お手軽JTAG-ICE

性能を評価したり，まれにしか出ないバグを追うといった目的には，名の通ったメーカの，それなりの機能をもったICEを購入する必要があります．しかしレジスタの値をチェックしながらステップ実行してみるという程度なら，もっと簡易的なJTAG-ICEの導入を検討するのもよいかもしれません．たとえば，筆者のお気に入りは，NetSilicon社のDigiConnect ME開発キットに含まれていた簡易JTAG-ICEです(写真A)．PCのパラレル・ポートに接続し，GDB経由でターゲットにアクセスできます．ネット通販でICE単体を買っても700ドル前後のようです．

また，国産でも平成15年度IPA未踏ソフトウェア創造事業の成果を利用した「包括的JTAGサポートソフトウェア」をベースとするJTAG-ICE環境が存在します．GDB経由でデバッグできる環境のほか，Windows版ARM7対応JTAG-ICEをリリースするなど，活発な開発を行っており，手頃なJTAG-ICE環境の一つとして要注目です．これに関してはhttp://www.nahitech.com/から情報を得ることができます．

これらは機能が質素ではあるもののコスト・パフォーマンスは高く，カーネルの移植においては十分に強力な戦力になります．



写真A　簡易JTAG-ICE



写真1　CQ RISC評価キット XScale

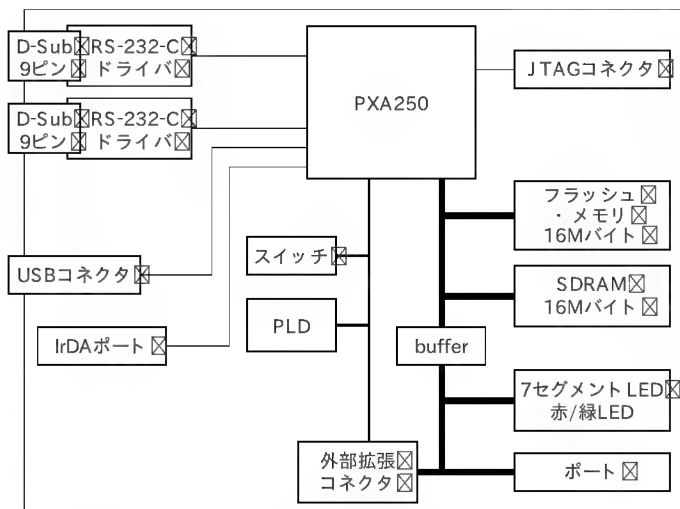


図3　CQ RISC評価キット/XScale CPUボードのブロック図

ます．

▶gdbstubを使うときの注意

ボード・サポート・パッケージにgdbstub(コラム3参照)が含まれていたとしても，それがそのままJSPカーネルの開発で使えるかどうかはわかりません．JSPカーネル用のgdbstubは，一般的に配られているものと微妙に挙動が異なっています．移植作業に入ると，JSPカーネルに意識が集中してしまって，よもやgdbstubの非互換性が原因とは思い至らないケースもあります．注意が必要です．

▶実作業時間

道具ではありませんが，集中できる時間も必要です．ターゲットの種類や移植者のスキルにもよりますが，JSPカーネルで対応済みのCPUの場合，特定ボードへの移植は，おおむね1週間程度が必要です．

今回のターゲット

さて，一般論はこれくらいにして，いよいよ移植ターゲットの選定に入ります．

●ARMv4に移植してみよう

目標は，JSP1.4付属のsample1アプリケーションをデバッガ経由で動作させるということにします．JSPカーネルを動作させるために絶対に必要なハードウェアのリソースは，「ms単位の精度を持つタイマ1個」と「ログ出力のためのシリアル出力」です．また，今回は移植入門編なので，GCCのサポートがあることを条件にします．sample1アプリケーションの動作のために必要なメモリは，かなり大きく見積もってもROMが40Kバイト程度，RAMは10Kバイトもあれば十分です．すべてRAM上で動作させても64Kバイト以内で収まります．

今回は移植のターゲットとしてARMv4を選択しました．その中で，入手が容易なものという理由で，CQ RISC評価キッ

コラム3　gdbstubとは?

GDBは，もともとはUNIX系のコミュニティで生まれ育ったデバッガです．

UNIXはマルチプロセス環境なので，基本的にGDBを動作させるマシンとデバック対象のプロセスを動作させるマシンが同一であることを想定しています．

しかし，UNIX系でもデバイス・ドライバやカーネルなどの開発では，ほかのマシンで動作しているプロセスのデバッグを行いたいという要求はあります．そこで，GDBにはメモリの内容やレジスタの状態をバイト・ストリームで送受信できるようなリモート・シリアル・プロトコルが定義されています．このプロトコルによってホスト側の要求に応じて応答を行うターゲット側のプログラムをスタブ(gdbstub)と呼びます．

組み込み分野では，ターゲット上にROMモニタを置きホスト上にデバッガのフロントエンドを置く，という方式がよく取られますが，大まかな方針はこれと同等なものです．

そこで，GDBを組み込みに用いる際には，ターゲットに合わせたgdbstubを作成しROMに焼いて，デバッグ・モニタとして利用するということがよく行われます．

さらにgdbstubを組み込み分野で発展させた例として，JTAG-ICEの実現があります．gdbstubに相当するプログラム(この場合gdbstubはターゲット以外のところにある)がパラレル・ポートなどを利用してJTAGをエミュレートし，MPUの情報をホスト上のGDBに渡すようにします．

コラム2で紹介したお手軽JTAG-ICEの中には，このようにしてフロントエンドをGDBに任せているものもあります．

コラム4 エミュレータで安価に始める

カーネルの勉強というと，ターゲット・ボード，デバッガ，シリアル・ケーブル…と初期投資が大きいのが悩みの種という話をよく耳にします．仕事に直結するのであれば，大した額ではないかもしれませんが，趣味の範囲として考えると，ちょっと痛い出費かもしれません．そういうときには，エミュレータの使用を検討する余地があります．

TOPPERSでエミュレータというと，WindowsやLinuxの上で動作するシミュレーション環境を思い浮かべる方が多いでしょう．実際，本連載でもWindowsエミュレータをベースとした解説が続きましたが，最近のPC環境の性能向上のおかげで，CPUそのものをエミュレートしてしまうタイプのエミュレータが実用的な速度で動作するようになっています．最近は，無償のオープンソース・ソフトウェアが増えたおかげでほとんど出費なく開発を開始できます．デバッガを経由してレジスタやデバイスを鷲づかみする感覚はWindowsやLinux上のエミュレーションでは体験しにくい醍醐味です．

実は，TOPPERSカーネルの開発メンバの中にも，エミュレータの愛用者がいます．もちろん開発メンバは動作する実機をいくつも持っていますが，飛行機の中でもコタツの中でも開発できることに大きなメリットを感じているようです．

たとえば，armv4ターゲットの担当である本田氏は，Armulatorでの開発を行っています．筆者も，Bochs(`http://www.bochs.org/`)やqemu(`http://www.qemu.org/`)で大方のバグを取ったうえで実機でのテストに入るようにしています．Bochsやqemuにはgdbstubのサポートがあり，ちょっとしたICEに負けないデバッグ環境が実現できます．

GDBそのものがCPUシミュレータを持っているターゲットもあります．JSP1.3まで存在していたV850ターゲットはgdbでの動作を前提としていました．また，TOPPERS/FI4カーネルの大部分は，パッチを当てたh8300-hms-gdbを使って開発され，後でms7727cp01ターゲットに移植されました．

オープンソースではないものの，無償で入手できる環境もあります．XStormy16ターゲットを三洋マイコン開発ツールで動作させる方法はJSP1.4のソース・アーカイブ中の`doc/xstormy16.txt`に書かれています．

今回は移植作業に関する話題ですので，今回は関連情報として挙げるにとどめますが，これらのエミュレータを使った開発についてはいつか別の機会に，解説したいと思っています．

トのXScaleをターゲットとしました(写真1，図3)．

XScaleは厳密にはARMv5互換で興味深い独自拡張をいろいろと付加していますが，今回は単に高速動作するARM7互換CPUとして扱います．また，ボードには，7セグメントLEDが一つ，LEDが二つ，6ビットのユーザ開放ディップ・スイッチ，二つのユーザ開放プッシュ・スイッチ，USBコネクタ，IrDAポート，2系統のRS-232-Cドライバなど豊富なデバイスが実装されていますが，ほとんど何も使いません．メモリもCS0の空間に16Mバイトのフラッシュ・メモリとSDRAMバンク0に16MバイトのPC100仕様のSDRAMがあります．

このキットには，GCCがバイナリで提供されており，ROMデバッガとして(株)ソフィアシステムズ社製WatchpointのCQ版が付属しています．Watchpoint自身は，JTAGやROM ICEにも対応している統合デバッグ環境ですが，CQ版ではROMモニタ経由のみに限定されているようです．

ボードにはJTAG端子もついているので，適切なJTAG-ICEを接続し，使い慣れたデバッガを利用することも可能ですが，今回は読者の方々が手軽に試せるようにするため，Watchpoint CQ版を用いることにします．

Watchpointは GCC専用のプロダクトではないため，GCCとWatchpointのバージョンによっては微妙な相性問題が発生することがあるようです．筆者も，特定のバージョンで，デバッグ・シンボルが正しく読み込めないという問題に遭遇したことがあります．今回は添付のGCCを使うため，このような問題が起こることを心配する必要はなさそうですが，ほかのバージョンのGCCを使う場合には注意が必要です．

●JSPカーネル移植の準備

まずは，実環境を整えます．ボード固有の事情などもいくつかあるので，少し細かく説明していきます．

▶開封と機材調達

早速，CQ RISC評価キットのXScaleを開封します．箱の中には，CD-ROMが1枚，ACアダプタとXScale搭載ボードが入っています．ARM純正コンパイラの評価版も含まれていますが，今回はGCCを使うので不要です．RS-232-CやUSBなどのケーブルは入っていないので別途用意してください．

以降，評価キットに含まれるXScale搭載ボードを，単にターゲットと呼びます．

今回の開発のためにWindows XP Professionalが動作しているノートPCを1台用意しました．このノートPC上で，カーネルのビルドを行い，デバッガ経由でターゲットにロードし，デバッグするというのが，開発の一連の流れになります．以後，このノートPCをホストと呼ぶことにします．今回用意したホストには，RS-232-Cシリアルのポートがありませんでした．そのため，USB-シリアル変換アダプタを一つ用意しました．

なお，余談ですが，USB-シリアル変換アダプタは，メーカによって，ドライバの完成度に大きな差があるようです．ターゲット側でボーレートを途中で変更したり，ターゲットのリセットを頻繁にかけるとWindowsがクラッシュする例もあります．具体的な社名を挙げることは避けますが，ネットなどで情報を集め，実績のあるものを選ぶようにすることを勧めます．

▶結線

機材が揃ったら早速ホストとターゲットを結線します(写真2)．

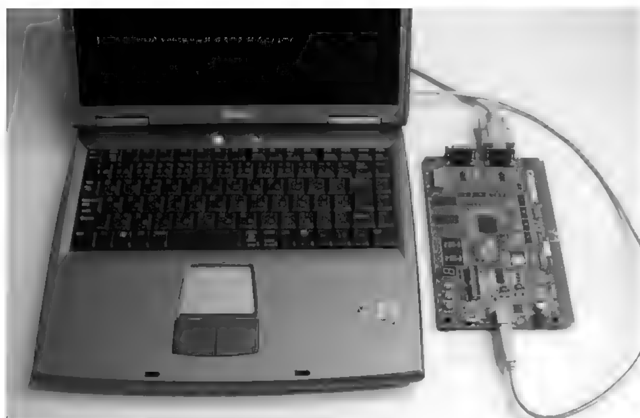


写真2　筆者が用意した環境

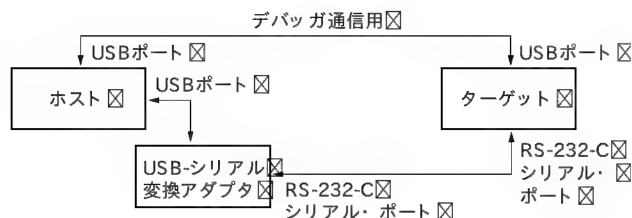


図4　結線図



図5
展開後のディレクトリのようす

```
C:¥Documents and Settings¥monaka.MONAKA-PC>arm-elf-gcc
arm-elf-gcc: No input files

C:¥Documents and Settings¥monaka.MONAKA-PC>
```

図6　コマンド・プロンプトの出力

ターゲットに乗っているモニタROMとデバッガの通信には，シリアルもしくはUSBのどちらかを選ぶことができます．しかし，USB-シリアル変換アダプタ経由では正しく動作しないケースがあるようです．

そこで，モニタROMとデバッガの通信にはUSBケーブルを使うことにします．これも，RS-232-Cシリアルが2ポート搭載されているホストの場合には，シリアルを使ってもかまいません．そういうPCは最近少なくなってしまいましたが….ちなみに，USBを使ってデバッガと通信する構成では，ターゲットにある二つめのシリアル・ポートが使えなくなる(PLDのアップデートで使用可能)そうです．今回の移植作業には影響ないのですが，応用製品を考えている場合には注意が必要です．

以上の結果をまとめると，図4のようになります．また，筆者が用意した実環境を写真2に示します．

▶ ツール類のインストール

結線が終わったら，ツール類のインストールに入ります．この手の評価キットはご多分に漏れず(？)，CD-ROMに含まれる文書にはソフトウェアのインストール手順に関しては必要最低限の情報しか書かれていません．

まずは，Watchpoint CQ版をインストールします．インストール方法は，CD-ROMのルート・ディレクトリではなく，`¥WP¥README.TXT`にあります．このファイルの記述にあるとおり，`¥WP¥DISK1¥SETUP.EXE`を実行すると，数回のクリックでインストールできます．最後に再起動するかどうかの確認があるので，念のため編集中のアプリケーションがないことを確認したうえで再起動します．

次にGCCをインストールします．GCCの一式は，`¥Tool¥arm-elf¥arm-elf.zip`で，ZIP形式でアーカイブされています．この中身を単純に展開すれば動作するはずなのですが，適当な位置に展開すると，思わぬところでトラブルに遭います．

まず，GCCはUNIX系OS由来のツールであるため，空白文字や日本語文字を含むようなディレクトリが関与すると，正常な動作はまず期待できません．GCCは，必要なディレクトリ情報をバイナリ中に埋め込むため，適当なパスに展開すると，思わぬところでトラブルに遭います．しかし，GCCをどこに展開先すればよいのかは明示されていないようです．

できるだけトラブルを避けたかったので関連するドキュメントを読んでみたところ，`¥WD¥PXA250CQ.PDF`の5.1.3に，ヒントとなる情報がありました．どうやら`c:¥tools`の下に展開するのが正解のようです(図5)．

このように展開したうえで，`PATH`環境変数に`c:¥tools¥armelf-981216¥H-i586-cygwin32¥bin`を追加します．`PATH`環境変数の設定方法は，NT系列(NT/2000/XP)と95系列(95/98/98SE/Me)で異なっており，TOPPERSの話題から離れすぎるので割愛します．

ここまでの設定が終わったら，コマンド・プロンプトを開き，`arm-elf-gcc`を実行してみます．`Arm-elf-gcc: No input files`という返答があれば，ここまでの設定は問題なくできています(図6)．

これでターゲットが用意してくれているソフトウェアのインストールは終了です．CD-ROMにあるサンプルを用いて動作させることができます．

しかし，JSPカーネルに関連する開発を行うためには，あといくつかのソフトウェアが必要になります．今回は誌面が尽きてしまったので，次回に，追加のソフトウェアのインストール作業を行い，JSPカーネルの移植作業を行います．

むらなか・まさき　(資)もなみソフトウェア

MMUなしでも動作するLinux

はじめて使うμClinux

大谷 浩司/高岡 正/近藤 政雄/臼田 尚志

最終回 MMUなしプロセッサ用Linuxの共有ライブラリ機構

1 はじめに

MMUがないプロセッサ向けのLinuxとしてμClinux[1]（uClinuxと表記されることもある）が開発されている．μClinuxでは，仮想記憶機構をサポートしないため，通常のLinuxに比べていくつかの制限が存在する．共有ライブラリ機構もその一つである．通常のLinuxの共有ライブラリ機構は，仮想記憶機構を利用して実現されているため，μClinuxでは動作しない．しかし，共有ライブラリはメモリ消費抑制やソフトウェア保守などに有効であり，実現の要望が強い．

筆者らは，このたびμClinux向けの共有ライブラリ機構を開発し，実装したので報告する．以下では，まず，μClinux上の共有ライブラリ機構の現状を述べた後，筆者らの開発した方式の利点を述べる．次に，方式の概要を説明し，続いて詳細を述べる．最後にほかの方式について説明する．

2 μClinux上の共有ライブラリ機構の現状

現在，μClinuxの共有ライブラリ機構はARMプロセッサとColdFireプロセッサにのみ存在する．ARMについてはRidgeRun社が2002年の3月に開発を発表したが，残念ながらRidgeRun社は活動を停止した．ARMの共有ライブラリ機構は，現在，Cadenux社[2]が受け継いでいる[3]．ColdFireについては，SnapGear社[4]が，2002年の4月に開発を発表し，提供している[5]．これらの方式については，後ほど説明する．

3 共有ライブラリ機構の利点

筆者らの開発したμClinuxの共有ライブラリ機構は，通常のLinuxのものの利点をほとんどすべて受け継いでいるため，以下の利点がある．

1) ライブラリ共有によりメモリ消費を抑制できる

一つのプログラムだけで考えれば，共有でない場合よりもメモリ消費は多くなる．しかし，共有する部分が多ければ，システム全体としてのメモリ消費は少なくなる．これは，多くのライブラリ関数を使用する複雑なプログラムほど効果が大きい．

2) プログラムのコード共有により，メモリ消費を抑制できる

プログラムのコードなども複数のプロセスで共有可能なため，シェルなど複数のプロセスが動作するプログラムの場合は，メモリ消費を抑制できる．

3) 実行オブジェクト・ファイルが小さくなり，ファイル容量を削減できる

各アプリケーションに重複して含まれていたライブラリのコードが各ファイルに必要でなくなるため，ファイル・システム全体としてファイル容量を大きく削減できる．

4) ライブラリを修正してもプログラムのリンクをやり直す必要がない

プログラム起動時に動的にリンクするため，ライブラリに変更があっても実行が可能である．このため，プログラムの再リンクが必要でなくなり，プログラムの保守が容易になる．また，プログラムのオブジェクトを変更せずに，ライブラリの特別バージョンを利用することもできる．

5) 通常のLinuxのものと比較して実行オブジェクト・ファイルが小さい

μClinuxで利用される実行ファイル形式はbFLTと呼ばれ，実行オブジェクト・ファイルが小さい．今回開発した方式は，bFLT形式を拡張した実行ファイル形式を採用し，通常のLinuxのELF形式に比べて，40%から15%ほど小さい．

6) XIP（eXecute In Place）に対応している

ロード時にコード部分を一切変更しないため，ROM化した場合などに，カーネル・イメージをRAM上へコピーせずにROM上のLinuxカーネルをそのまま実行させることが可能である．これは，メモリ消費を大きく抑制する効果がある．

4 今回開発した方式の概要

本方式の理解を容易にするために，まず，既存のμClinuxでのプロセスのメモリ上のイメージを説明する．

図1に示すように，イメージは，textセグメント，dataセグメント，stackセグメントの三つのセグメントから構成される．

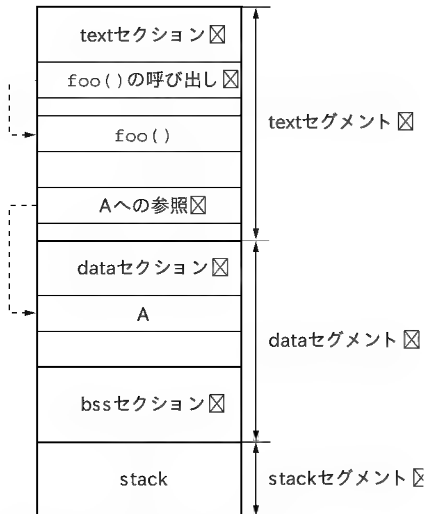


図1　プロセスのメモリ上のイメージ

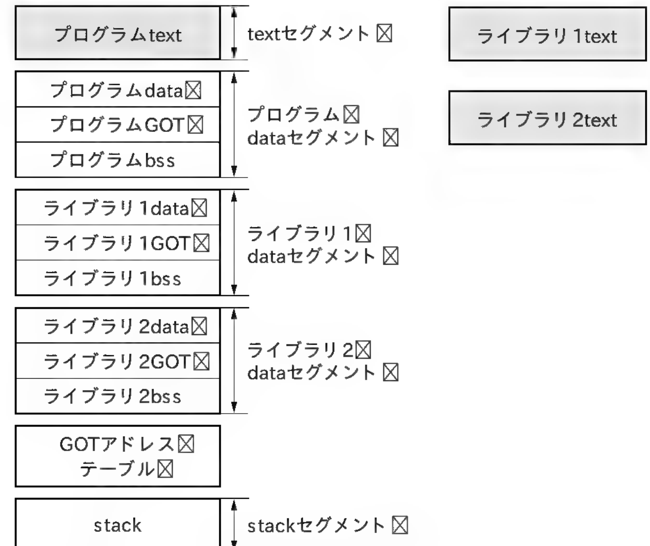


図2　共有ライブラリ機構のメモリ上のイメージ

textセグメントには，コードや書き換え不可のデータが入るtextセクションが含まれる．dataセグメントには，明示的に初期化されたデータの入るdataセクション，明示的に初期化されていないデータが入るbssセクションが含まれる．stackセグメントには，スタックが割り当てられる．通常のLinuxとは異なり，スタックはプロセスごとに固定サイズが割り当てられ，拡大することはない．

text，data，stackの各セグメントは，この順にメモリ上に連続して割り当てられる．コードはPIC(Position Independent Code)になっている．関数呼び出し，データ参照はともに，セグメントがメモリ上で連続していることを前提にPC(Program Counter)相対で行われる．

次に，共有ライブラリ機構でのプロセスのメモリ・イメージを図2に示す．

図2は，プログラムが共有ライブラリ1と共有ライブラリ2を利用する場合を示している．

共有ライブラリ機構では，textセグメントを各プロセスで共有し，dataセグメントを各プロセス固有のものを割り当てる．プログラムと各ライブラリのtextセグメントは，メモリ上で独立に存在する．プロセスを起動したときに，そのプロセス用のdataセグメントとstackセグメントが確保される．dataセグメントは，さらにプログラムと各ライブラリ用のdataセグメントに分かれている．

プログラムまたはライブラリ(以降は，単にモジュールと記す)用の各dataセグメントには，GOT(Global Offset Table)セクションが追加されている．ほかのモジュールへの参照は，GOTのエントリを介した間接参照になっている．

また，各モジュールのdataセグメントの次には，GOTアドレス・テーブルが存在する．GOTアドレス・テーブルは，各モジュールが自身のdataセグメントのアドレスを知るために使用される．

各モジュールでは，自身のdataセグメントをベース・レジスタ相対でアクセスしている．各ベース・レジスタは，呼び出された関数の入口でGOTアドレス・テーブルを利用してGOTの先頭を指すように設定される．

次項では，このイメージを利用していかに共有ライブラリが動作するかを詳細に述べる．

5 今回開発した方式の詳細

● GOT(Global Offset Table)

共有ライブラリなどで，textセグメントを書き換えないで動的にリンクする場合には，GOTという手法が知られている[6]．本方式でもGOTを採用している．

共有ライブラリにおいて，ほかのモジュールの変数や関数などのオブジェクトを参照する場合，そのアドレスは，モジュールを作成したときには未定である．そのため，プログラムのコードから直接参照すると，ロード時にコードを変更する必要が出てくる．しかし，本方式のようにROM化を許したり，共有したりする場合にはtextセグメントを変更することはできない．そのため，ほかのモジュールのオブジェクトを参照する場合は，自身のdataセグメントを介して間接的に参照する．この間接参照を行うためのテーブルをGOT(Global Offset Table)と呼ぶ．GOTの各エントリには，ほかのモジュールのオブジェクトのアドレスが入る．GOTのエントリに正しいアドレスを設定するのは，動的リンカ&ローダの責任である．

図3にGOTを利用した参照の例を示す．図3では，プログラムは共有ライブラリ1を利用している．プログラム側に変数

A，ライブラリ 1側に変数 B，関数 foo()が存在している．各dataセグメントに，GOT を設ける．プログラムから，ライブラリ 1の変数 B を参照する場合には，プログラムの GOT のエントリを介して間接参照する．また，プログラムからライブラリの関数を呼ぶ出す場合にも，GOT からそのアドレスを取得した後に呼び出す．ライブラリ 1からプログラムの変数 A を参照する場合には，ライブラリ 1の GOT のエントリを介して間接参照する．

● data セグメントのアクセス方式

text セグメントと data セグメントは，メモリ上で分離され，その配置はロード時になるまで決定されない．また，text セグメントを書き換えてはいけない．そのため，コードからは絶対アドレスを使って data セグメントをアクセスすることはできない．さらに，text セグメントと data セグメントの相対的な位置もロード時になるまで決定されないため，PC 相対でアクセスすることもできない．

そこで，data セグメントを指すベース・レジスタを設けて，このベース・レジスタ相対で data セグメントをアクセスすることにした．このベース・レジスタは，data セグメントのどこを指していても良いのだが，中心のあたりを指すほうが正負両方のオフセットが利用でき，より多くのデータのアクセスが容易になる．本方式では GOT セクションを data セグメントの中心部に配置し，GOT セクションの先頭を指すこととした．このベース・レジスタを GOT レジスタと呼ぶことにする．data セクションは負のオフセット，bss セクションは正のオフセットを使ってアクセスする．

data セグメントをアクセスする場合のオフセットは，モジュール作成時に決定されていなければいけない．共有ライブラリでは，実行時にどのようなモジュールとの組み合せで利用されるかは不明である．そのため，ただひとつの data セグメントを割り当てる方法は，オフセットが決定できないために，採用できない．そこで，各モジュールごとに data セグメントを用意し，各モジュールを実行する場合には，図 4 に示すように，そのモジュールの data セグメントを指すように GOT レジスタを設定する．

ここで，問題となるのが，GOT レジスタにいつどのようにして正しい値を設定するかである．次項では，この問題と本方式で採用した解決法について説明する．

● GOT レジスタの設定方法

まず，プログラムの GOT レジスタは，メモリ配置を知ることが可能な動的リンカ&ローダが設定する．GOT レジスタの変更が必要になるのは，ほかのモジュールの関数を呼び出すときである．設定方法は，大きく分けて，呼び出すモジュールで行う方法と，呼び出されたモジュールで行う方法の二つある．

呼び出すモジュールで行う方法には，次の欠点がある．

関数のポインタをほかのモジュールに渡した場合を考える．ほかのモジュールは，そのポインタを利用して関数を呼び出す．しかし，呼び出す場所では，そのポインタがどのモジュールの関数を指すポインタかを区別する手段がない．したがって，GOT レジスタを正しく設定することができない．そのため，呼び出すモジュールで設定する方法では，関数へのポインタをほかのモジュールに渡すことができないのである．

筆者らは，呼び出されたモジュールで設定する方法を採用している．実際には，関数の入口で GOT レジスタを設定する．これは，ほかのモジュールから呼ばれる関数だけで良い．しかし，関数のポインタのことを考えると，その関数がほかのモジュールから呼ばれるかどうかは簡単にはわからない．そこで，すべての data セグメントをアクセスする関数で GOT レジスタの設定を行っている．

次に，自身の data セグメントのアドレスを知る方法が問題となる．以下に，本方式で採用した GOT レジスタの設定方法を述べる．

まず，共有ライブラリに 1から始まる番号を付ける．この番

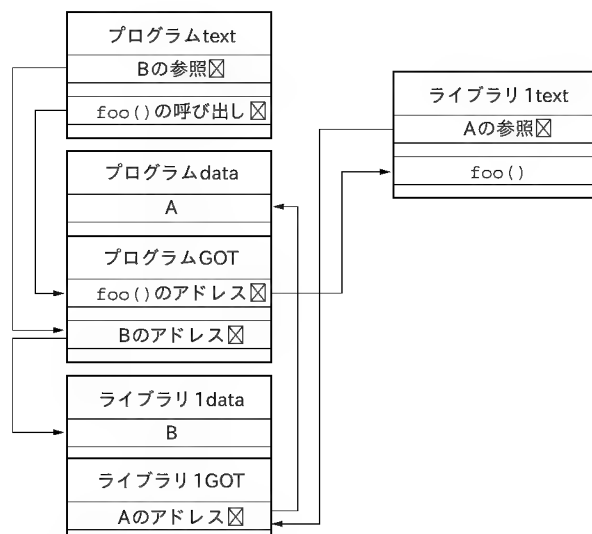


図3　GOTを使った参照

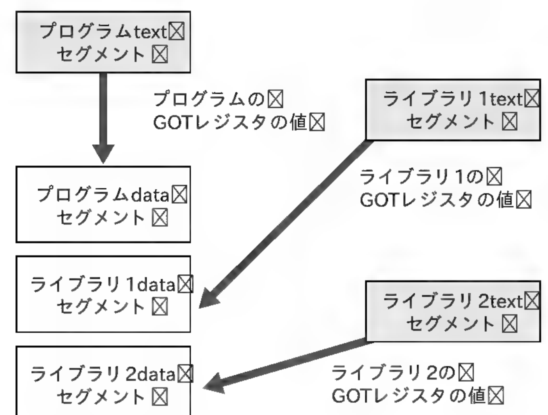


図4　GOTレジスタ

号をライブラリ番号と呼ぶ．プログラム本体には，番号0を割り当てる．ライブラリ番号順にモジュールのGOTの先頭アドレスを並べたテーブルをdataセグメントに用意する．これをGOTアドレス・テーブルと呼ぶ．各モジュールのGOTの先頭のエントリには，GOTアドレス・テーブルの先頭のアドレスを入れておく．図5に例を示す．この例では，共有ライブラリ1と共有ライブラリ2を利用しており，それぞれライブラリ番号1と2が割り当てられている．

このようにすると，自身のライブラリ番号を知っていれば，自身のGOTの先頭アドレスは，次の手順で簡単に得られる．

1) 現在のGOTレジスタの指すエントリから，GOTアドレス・テーブルの先頭アドレスを取得する．すべてのGOTの先頭エントリには，GOTアドレス・テーブルの先頭アドレスが入っているので，どのモジュールから呼ばれたとしても正しい値が得られる
2) 自身のライブラリ番号を n とすると，GOTアドレス・テーブルの n 番目のエントリには，自身のGOTの先頭のアドレスが入っている．これをGOTレジスタに設定すれば良い

GOTアドレス・テーブルのエントリの衝突を防ぐために，共有ライブラリのライブラリ番号はシステムでユニークな値でなければならない．本方式では，コードの簡単さ，高速性を考えて，共有ライブラリ生成時にユニークな番号をコード中に埋め込むことにした．しかし，プログラムが利用していない共有ライブラリのGOTアドレス・テーブルのエントリは空欄となるため，むだが生じる．とはいえ，このむだは，かりに100エントリあっても400バイトであり，許容できるものであると筆者らは考える．

GOTアドレス・テーブルを作成し，エントリを設定するのは動的リンカ&ローダの責任である．

● 動的リンカ&ローダの動作

動的リンカ&ローダは，以下の手順でプログラムをロードする．

1) ファイルからプログラムのtextセグメントをメモリ上にロードする．すでにメモリ上に存在すれば，それをそのまま利用する
2) プログラムのdataセグメントをメモリ上に確保し，dataセクション，bssセクションを初期化する
3) プログラムが参照しているライブラリのtextセグメントをメモリ上にロードする．すでにメモリ上に存在すれば，それをそのまま利用する
4) プログラムが参照しているライブラリのdataセグメントをメモリ上に確保し，dataセクション，bssセクションを初期化する．ただし，このプロセス用にすでに，このライブラリのdataセグメントが作成されていれば，作成しない
5) ライブラリが参照するライブラリについても同様に，textセグメントのロード，dataセグメントの作成を行う．これは，連鎖的に参照しているライブラリがすべてメモリ上にロードされるまで繰り返す
6) GOTアドレス・テーブルをメモリ上に確保する
7) 以下を各モジュールについて行う
   7-1) GOTの先頭アドレスをGOTアドレス・テーブルの対応エントリに設定
   7-2) GOTの初期化
   7-3) リロケーション
   7-4) dataセクションおよびGOTセクションのインポート・シンボルを解決
8) スタックをメモリ上に確保し，初期化
9) プログラムのGOTレジスタを設定して，プログラムを実行開始

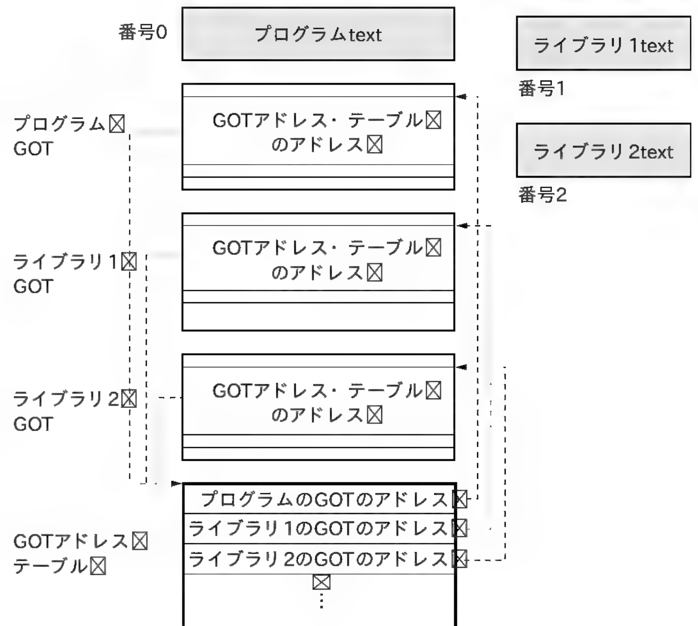


図5 GOTアドレス・テーブル

この手順のなかで，textセグメントをロードする部分では，すでにロードしている場合には，それを共有して利用する必要がある．通常のLinuxにおいては，この処理は，共有属性を指定してカーネルのmmapを呼び出し，ファイルをメモリ上にマップする（以降は，便宜上，共有mmapと記す）ことで実現できる．しかし，μClinuxでは，ファイルの共有mmapは一般には，実装されていない．そのため，本方式では，IPC（Inter Process Comunication）の共有メモリを改造して実現している．

● XIP（eXecute In Place）

ROMやRAMなどメモリ上のファイル・システムでのmmapが直接そのメモリ上のポインタを返すように実装されている場合は，XIPが実現できる．実験的にromfsに実装してみたところ，正常に動作することが確認できた．

● 初期化ルーチンの問題

C++などの言語では，`main`ルーチンを実行する前に，初期化ルーチンを実行する必要がある．これは，各ライブラリでも

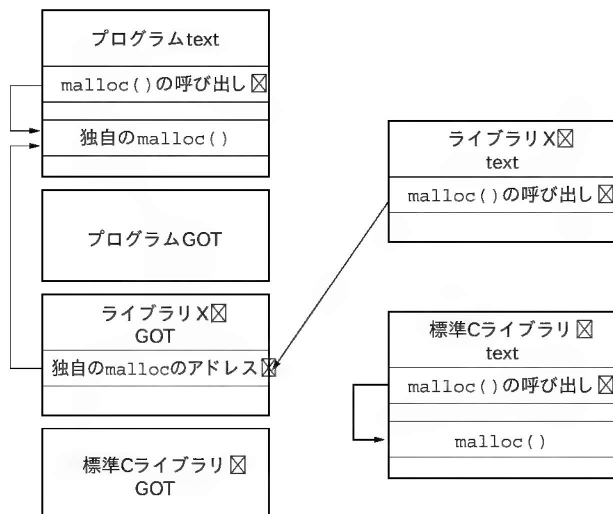


図6　関数の置き換え

必要である．

共有ライブラリでない場合は，すべての初期化ルーチンは，CTORセクションにエントリ・アドレスが集められ，リストになっている．このリストに基づいて，プログラムの__mainから呼び出される．

しかし，このままでは，共有ライブラリの場合，ライブラリ側の初期化ルーチンはプログラムの初期化ルーチン・リストに入らないため，初期化が行われない．

そこで，本方式では以下のように，この問題を解決している．

1) ライブラリ・モジュールにも__mainに相当するルーチンを設ける． これをinitializerと呼ぶことにする．このルーチンでは，そのライブラリの初期化ルーチンをリストに基づいて呼び出す
2) 初期化ルーチン・リストに，そのモジュールが参照しているライブラリのinitializerのアドレスを加える．これにより，プログラムの__mainから連鎖的にすべての共有ライブラリの初期化ルーチンが実行されることになる
3) 複数のモジュールから参照されているライブラリのinitializerは，複数回呼ばれるので，initializer内部では，実際の処理は1度のみ実行するようにする

● 関数の置き換えの問題

すでにライブラリ中で定義されている関数を独自のもので置き換えたい場合がある．たとえば，標準Cライブラリで定義されているmallocを独自のアルゴリズムのものに置き換えたい場合などである．

しかし，本方式では，モジュール内の関数呼び出しは，GOTを介さずに直接行われるため，置き換えることができない．たとえば，標準Cライブラリとライブラリ Xを利用したプログラムがあり，プログラム中に独自のmallocを定義したとすると，図6のようになる．

図6に示すように，プログラムとライブラリ Xのmallocの呼び出しは，プログラム中の独自のmallocを呼び出すが，標準Cライブラリ中の呼び出しは，標準のmallocのままである．

この問題を解決するために，本方式では，ライブラリ作成時のオプションによってすべての外部リンクの関数の呼び出しをGOT経由にすることができるようにした．このオプションを指定すると，同一モジュール内の関数であっても外部リンクの関数であれば，GOT経由の呼び出しとなる．この結果，例のプログラムでは，標準Cライブラリ内のmallocの呼び出しもGOT経由でプログラム・モジュールの独自のmallocを呼び出す．ただし，このオプションを指定すると，実行オブジェクト・ファイル中のシンボルの増加，モジュール内の呼び出しのオーバヘッドの増加となる．

● 既存ライブラリの利用

本方式では，プログラム，ライブラリともに，本方式用にコンパイルするのが原則である．しかし，この方法は，すでに手元にライブラリがあって利用したいがソースは持っていないような場合に問題となる．本方式では，制限を設けてこのようなライブラリをプログラムに静的にリンクすることを許した．

このようなプログラムのdataセグメントをtextセグメントに連続してメモリ上に配置することにより実現している．このように配置すれば，プログラムにリンクされた既存のコードから見ると，dataセグメントが既存のメモリ・イメージと同様に見えることを利用している．

既存のライブラリを静的にリンクしたプログラムは，共有ライブラリを利用することはできるが，プログラムのtextセグメントを共有することはできない．また，既存のライブラリは，共有ライブラリのデータをアクセスしてはいけない．なお，共有ライブラリに既存のライブラリをリンクすることはできない．

6 ほかの方式について

● Cadenux社の方式

Cadenux社の方式の情報は，同社のWebサイト(3)から得られる．

Cadenux社の方式は，最初は実行ファイル形式としてELFを採用していたが，後にbFLTを拡張した形式に変更した．この形式をxFLT注1と呼び，方式をXFLATと呼んでいる．

筆者らの方式と大きく異なっている点は，GOTレジスタに相当するSB(Static Base)レジスタの設定を呼び出し側のモジュールで行っていることである．そのため，ライブラリ番号は必要ないが，すでに指摘したように関数ポインタをほかのモジュールに渡す場合に問題が生じる．この問題の解決法として，Cadenux社はそのためのマクロと関数を提供している．これら

注1：筆者らの方式でも実行ファイル形式をxFLTと呼んでいる．偶然に名前が同じになってしまったが，まったくの別物である．

を使って，ソースの関数ポインタを扱う部分の変更を行う必要がある．また，Linuxのシグナルの処理についても，関数ポインタを扱うために，カーネルの変更が必要である．

また，textセグメントの共有では，共有mmapに頼っており，これを実装していないファイル・システムでは，共有することはできない．

● SnapGear社の方式

SnapGear社の方式の情報は，同社のWebサイトに存在するTechnicalBulletin(5)から得られる．また，同社は，共有ライブラリ機構をμClinuxのコミュニティに提供しており，μClinuxのWebサイト(1)からダウンロードできる．

SnapGear社の方式の特徴は，リンクを静的に行うことである．実行時には，ロードとリロケーションのみ行う．実行ファイル形式として，bFLTを採用しておりシンボルを一切含まない．

共有ライブラリには番号が与えられる．静的リンク時に，参照部分には仮の参照先アドレスが設定される．どのライブラリへの参照かを示すために，仮のアドレスの上位8ビットにライブラリ番号が入る．下位24ビットは，参照先の先頭からのオフセットである．ローダは，このライブラリ番号を見て，実際にライブラリがロードされたアドレスをオフセットに加えれば，実際のアドレスが得られる．ライブラリの名前は，ライブラリ番号から作成されており，ライブラリのロードもライブラリ番号がわかれば，可能である．

この方式では，実行オブジェクト・ファイルが小さくなり，ロード時の負荷も小さいという利点があるが，以下のような欠点を持っている．

1) ライブラリを変更すると，それを利用しているプログラムすべての再リンクが必要となる
2) いったんリンクするとライブラリ関数の置き換えは，一切できない
3) モジュールのアドレス空間が16Mバイトしかない
4) ライブラリをシステム内で最大255個しか利用できない

GOTレジスタに相当するレジスタへの値の設定方式は，筆者らのものと非常に似ている．しかし，GOTアドレス・テーブルは，プロセスに一つではなく，各モジュールごとに一つ存在する．そのため，GOTアドレス・テーブルのアドレスを求めるのに，メモリをアクセスする必要がない．したがって，メモリ・アクセスが1回減るが，消費メモリが増加するという欠点がある．

textセグメントの共有では，やはり，共有mmapに頼っており，これを実装していないファイル・システムでは，共有することはできない．

おわりに

本稿では，筆者らが開発したμClinux向けの共有ライブラリ機構について，その利点，方式について説明し，ほかの方式との比較を行った．筆者らは，すでに実際のプロセッサに実装し，有効性を確認した．XIPについては，本稿執筆時には，実験的に実装して動作を確認したが，今後，本格的に実装していきたい．

最後になったが，情報提供や，テストなど開発に協力していただいた方々に感謝したい．

参考文献，URL
(1) μClinux Webサイト，http://www.uclinux.org/
(2) Cadenux社Webサイト，http://www.cadenaux.com/
(3) Cadenux XFLAT Shared Libraries，http://www.cadenaux.com/xflat/
(4) SnapGear社Webサイト，http://www.snapgear.com/
(5) SnapGear Technical Bulletin #9，uClinux-Shared Libraries，http://www.snapgear.com/tb20020409.html
(6) John R.Levine著，榊原一矢 監訳，ポジティブエッジ訳；Linkers&Loaders，オーム社開発局

おおたに・こうじ/たかおか・ただし/こんどう・まさお/うすだ・なおし （株）アックス

シニアエンジニアの技術草子 四拾四之段

◆中秋の名月

旭 征佑

● 古代インドにさかのぼる，月とうさぎの関係

「月」と「うさぎ」にかかわるもっとも古い記録は，古代インドにあるようだ．広辞苑で有名な新村出博士によれば，古代インドのサンスクリット語では，月のことを「うさぎ」と発音するらしい．また，古代インドの「ジャータガ神話」にも月とうさぎに関する有名な話が登場する．

むかしむかし，「きつね」と「さる」と「うさぎ」がいた．彼らは「前世の行いが悪かったため獣になってしまった．今からでも何か人のためになろう」と常日頃そう思っていたという．これを聞いた帝釈天が，「何か良いことをさせてあげよう」と，行き倒れの老人に姿を変えて彼らの前にあらわれた．この老人を助けるために，さるはさっそく木に登って木の実や果物をとってきた．きつねは野山を駆け巡り，川から魚をとってきた．ところがうさぎは，できることが何もなかった．考え抜いたうさぎは，火の中に飛び込んで「どうぞ私の肉を食べてください」と，黒焦げになってしまった．これを見た老人は，帝釈天の姿に戻り，「りっぱな心がけだ」と感心し，その黒焦げになった姿を永遠に月に残し称えることにしたという．

この神話は，インドから戻ったばかりの玄奘（三蔵法師）が，西暦645年に「大唐西域記」で紹介している．もともと中国には，ある男の妻が不老不死の薬を盗んで月に逃げて「蛙」になり，月で薬をつくという伝説があったらしい．しかし，仏教が普及しだした7世紀ごろを境に，「うさぎ」が月で不老不死の薬をつくということに変わってしまったようだ．それほどまでに玄奘が広めた仏教の影響が強かったのかもしれない．

わずか20年ほど後の西暦662年．日本では，聖徳太子の逝去をしのんで推古天皇が天寿国曼荼羅帳（でんじゅこくまんだらちょう）（国宝）を作らせた．この刺繍の左上の隅には月が，右にはうさぎが書かれ，薬壷をついている．この話は，太子が広めた仏教とともに，飛鳥時代の日本にも伝わってきていたのだ．

平安時代の末期（1120年ごろ）に編纂（へんさん）された「今昔物語集」では，第五巻第十三話でジャータガ神話のうさぎの話が，ほぼそのままの形で掲載されている．当時すでに，日本の昔話になってしまっていたようだ．

一方で，うさぎが餅をつくのは日本独自のものと考えてまちがいない．「薬壷をつく」という姿が日本伝統の餅つきの姿に似ていたとか，藤原道長が詠ったように，満月は古来より「望月（もちづき）」ともいうが，これが「餅つき」になまったという説もある．

何より，実際に月を見ると，確かに餅つきをしているように見えるというのが正しいだろう．日本の習慣がそう見えさせているに違いない．

● 子供には見えない

筆者が小学校に入ったばかりのころ，月に関して変わった悩みがあった．実は月にうさぎが見えなかったのだ．じっくり見ると，うさぎに見えないこともない．このぐらいの年のころは，友人よりも大人のほうが影響が強い．周りにいる大人で月にうさぎが見えることはだれ一人として否定しなかった．結局，心の中でひっそりと月にうさぎが見えることに疑問を感じていた自分を覚えている．

たぶん，1993年6月4日のことだと思う．筆者は近所の仲のよかった幼稚園児2～3人と，皆既月食を待っていた．このとき何とはなしに「月にうさぎが見える？」と問いただした記憶がある．しかし，だれからもはっきりした回答が返ってこず，自分の子供のときの記憶がよみがえっていた．読者の皆さんは，小さかったころ，はたして月にうさぎが見えていたのだろうか？

なぜ月にうさぎが見える，などと思うようになったのだろうか．冒頭に説明した話など知る由もなく，満月に餅をついているうさぎを見た大人の表現が，子供に伝わったのかもしれない．母親が御伽噺（おとぎばなし）的に話したのかもしれない．そして，いつの間にか月にうさぎが見えるようになってしまった．本当に月にうさぎが住んでいると思い込んでいた人もいるかもしれない．

最近は月を見上げると，不思議なことに一瞬にしてうさぎが見える．この違いはもちろん，筆者が世間並みの風流人になったわけではないだろうし，数十年で月の表情ががらりと変わったわけでもないだろう．とすれば，大人になって，月にうさぎが見えるという先入観が筆者の中で完成されてしまったに相違ない．今は何度月を見ても，うさぎ以外のものが見えない．困ったものだ．

● ニューロンとパターン認識

哺乳類の発達した脳には，側頭葉連合野というところがあり，高度な聴覚認知，視覚認知をつかさどる．ここには「手ニューロン」があることが，1984年にハーバード大学のロバート・デ

シニアエンジニアの技術草子



シモン氏により報告された．リアルな手の形や，動きに際めて強く反応する．次いで「顔ニューロン」も報告された．顔ニューロンは，人を判別し，表情を一瞬で理解する．ほかにも，紙などを引き破ったときだけに反応したり，毛の生えた毛皮を手でなでたときにのみ反応したり，人が向かってくるときだけ反応するニューロンが見つかっているらしい．これらの多くニューロンがあるという説を『認識ニューロン仮説』という．そうすると，人間はそれぞれの感覚に対応して無数のニューロンが必要なことになってしまう．

しかし，現在の研究では，この考えは必ずしも正しくないということに落ち着きつつあるようだ．ある範囲の複数のニューロンが反応し，情報伝達物質の種類，相互の情報交換，経験・記憶の介在など多くの情報が相互に影響しあって動的にパターンを認識している可能性が高いという．

子供のころは，このパターン認識能力があるにもかかわらず，経験が少ないため，情報を最終的に判断することができない．何か得体の知れないものを見たり，聞いたりした気がする理由のひとつだろう．あるパターンを見たときの判断に，多くの可能性を残しているのだ．子供は，小さいときは月にうさぎは見えない．しかしある程度大きくなると，今度は本当に月にうさぎがいるように見えてくる．なかには月にうさぎがいることを信じたりもする．そして，大人になると，「確かに月にうさぎの影は見えるが，月にうさぎはいない」という自分では証明すらできない，しかも画一的な結論に達するのである．

パターン認識が完全に画一化している．悪くいえば，いわゆる石頭になってしまったということもできる．

● 月を見上げて

生産性を追及する場合に重要視されるのは，パターン化だ．実際，ほとんどの技術者は，仕事をパターン化し，生産性を上げ，納期を守ることで評価される．しかし，同時に失われるのは想像力や独創性でもある．仕事では，生産性を上げないとリストラされてしてしまう世の中だが，同時に独創性を失うと将来の設計図が狂うかもしれないのも事実だ．

会社で垂涎(すいえん)の的となる優秀な技術者ほど，この罠にはまる可能性が高い．しかも気づくとすれば40代になってからだから，たちが悪い．



中秋のころに月見をする習慣は古く，今をさかのぼる約3000年前，周の皇帝が月を祭った旧暦の8月15日の中秋節が始まりらしい．後に唐の皇帝玄宗が，中秋節の日に大がかりな月見の宴を毎年のように開くようになり，広く一般にも広まったという．遣唐使がもっとも盛んな平安時代に，日本にも伝わり，中秋に月見の宴を開くのが貴族のたしなみの一つになった．この時代には中国から月とうさぎに関する話も同時にいろいろ入ってきている．平安の殿上人たちは，和歌を詠み酒を嗜みながら，月にいるうさぎに想いを偲ばせたに違いない．

さて，あなたは，見上げた月に果たしてうさぎが餅をついて見えるだろうか．見えるとすれば，すでにパターン化された世界のなかにいることに改めて気付かなくてはいけない．夜遅く帰ったときは，月を見上げて，独創性を失いかけているかもしれない自分にときどき警告を発することを忘れないようにしてほしい．今年の中秋は，9月28日だという．この日に見える満月が，中秋の名月だ．

あさひ・しょうすけ　テクニカル・ライタ
イラスト　森 祐子

電脳事情にしひがし

猪飼 國夫

国内外に見る研究学園都市とハイテク産業の集中化…中国編（下）

今回は，前回に引き続きハイテク産業の誘致をめざすケース・スタディとして，中国の地方都市を例に取り報告します．

● 中国安徽省・合肥市

中国では故鄧小平氏が主導して大連，北京，天津，上海，広州など沿岸部が高度に発展しています（図1）．それに対して内陸部の省は，農業や鉱業などの第一次産業を主体としています．そのため，各省の省都にある有名大学を出ても地元での有力な就職先は少ないといわれています．



図1　中国のおもな都市



図2　合肥サイエンス・パーク計画図

合肥（がっぴ）市は華北平原（中原の地）と江南の接点に当たる歴史的に要衝の地で，華東地区の安徽省の省都です（人口450万人）．三国志のころから南征北伐の軍隊がこの地を蹂躙していたため，観光資源としては合戦の場や李鴻章の故居などいくつかあるだけで，近代産業としてはこれといったものは見当たりません．

この街をハイテク基地化しようというのが，省や市当局の考えのようです．安徽省は南京のすぐ西に広がる南北500km，東西300kmほどの人口5,700万人の省で，省都との関係もインドのKarnataka州とBangaloreに似ています．

● 高新技術産業開発区とその周辺の概要

省政府の計画によると，1991年に国家計画に採択されてから旧市街の外10kmの南西部に大学，中国科学院分院，科学城（サイエンス・パーク）および区域外の経済技術開発区（工業団地）などからなる，世界に伍せる規模のハイテク基地を作りつつあります（図2）．

このハイテク基地でやろうとしていることは，

1) ソフトウェア作成の国外・国内からの受注
2) データ打ち込みなどの作業の国外からの受注
3) 独自ソフトウェアの開発と販売
4) ICの設計と開発の受注
5) バイオ関係の開発と薬品などの製造・販売

などが，おもな方向のようです．

教育研究機関としては，合肥市には日本の大学校に相当する中国科学技術院傘下の中国科学技術大学があります（写真1）．この大学は理系では中国で三本の指に入るという有名大学ですが，清華大学や上海交通大学のように沿岸部にないことと，研究内容や卒業生の対外進出があまりないためか，外国ではあまり名が知られていません．その他，合肥工業大学など多くの高等教育機関から供給される人材や研究成果を基にして産学協同方式でのハイテク基地化を考えているようです．

工業団地で現在操業している企業は日立建機（80%出資）や三



写真1
中国科学技術大学の情報科学技術大学院

洋電機(半額合弁)など外国企業群と安価な家電製品で世界有数の大企業になった中国の海尔(Haier)など，安価で優秀な労働力を求めた既成の大企業が目立ちます．

サイエンス・パークには半導体産業の基地とハイテク企業のインキュベーション用に貸し出されるソフトウェア，バイオ，北米留学経験者などの創業を手助けするKSP(神奈川サイエンス・パーク)のようなビル群があり，いくつかの中小システム・ハウスが，大学からの技術や日本など国外のソフトウェア受注をこなしています．

● 技術者とことばの状況

技術者の給料ベースは，大学院修士課程卒業程度の技術者で月額5,000元(1元13～14円)程度で，上海や北京の半分くらいです．その分生活費も安く，郊外のマンションは北京や上海の1/3程度の費用で入手でき，食事や交通にかかる費用も少なくて済むようです．

有名な大学の卒業生の学力水準は非常に高く，英語で技術の会話が成り立つ人もたくさんいます．ただ，プライドが非常に高いため，日系企業に就職したり日本人と交わるよりは，欧米系の企業や欧米人と交わりたいという気持ちの学生が多いように見受けます．

しかし，大学を卒業しても現在の状況ではこのハイテク基地での雇用需要はあまり多くなく，進出する企業にとっては買い手市場です．役所の若い人に博士の学位を取得した人が非常に多く，以前中国で名刺に盛んに書かれ，教授相当と説明されていた「高級工程師」の代わりに使われているようでした．博士の学位も日本よりは大量に発行されていて，就職先が少ない中で役所を選んだという印象を受けました(写真2，写真3)．

外国語は英語が主ですが，観光地ではないせいか，街ではほとんど通じません．大学生と国際関係の人だけが英語で話ができるという状況です．この点がBangaloreやPhilippineなどとは大きく事情が異なります．

日本語は，元留学生・就学生および現地の日系企業での就業者と日本から仕事を受注している企業を主に，学んでいる人はある程度います．中国語ができない人は，仕事以外では彼らに通訳を頼まないと街での暮らしはたいへんなようです．

中国語は，北方方言(北京語など)の地域の南端に属していて，上海や南京のように土地の人のことば(上海語など)と標準語(普通話)の両刀使いではないようです．

● 街の印象

合肥市は省都とはいえ，中国では田舎の都市という印象が拭えませんでした．城壁の跡の公園に囲まれた旧市街は，どこにでもある中国の都市です．中国の都市は歴史的には政治・軍事を中心に成り立っています．おもな産業は商業というのが普通



写真2　情報産業庁の幹部(右：賀凌庁長，左：王洵博士)



写真3
ハイテク開発区経済貿易局の副局長，李紅博士

でした．観光都市ではないこともあり，中国の大都市ではめずらしく安全な街であるという感じでした．

実際に南京からの距離は150kmしかなく，上海まで400kmほどの高速道路も開通しているので，時速100kmなら4時間で行き着くことができるわけですが，移動する人が少ないせいか飛行機や鉄道，高速バスの便はあまり整備されていないようです．たとえば，上海からの飛行機は国際便が発着する浦東空港からではなく，以前の国際空港の虹橋空港からで，二つの空港の間の移動は，成田-羽田や関空-大阪のように上海市内を横断しなければなりません．

● 合肥市と日本との関係が築けるか

この街と日本との関係をいくつか考えてみます．いくつかの場合が想定できます．

1) 従来型の製造業の進出
2) データ打ち込みやコーディングなど簡単な情報産業の進出や外注
3) IC設計やソフトウェアなどシステム開発産業の進出や外注
4) 個人的な就職先

項目の番号が増えるに従ってより発展したハイテク基地と考えられます．

● 従来型製造業の進出

この方面では，現地需要が大きい産業は成功すると考えてよいと思います．すなわち，沿岸部より安価で優秀な労働力が確保でき，電力や水，住宅などの産業基盤整備も広い土地を背景に進んでいるので，中国の事情に詳しければ日立建機のように同社では世界有数の工場となっているようです．

工賃も現在は2,000元未満で沿岸部より安く，地元の人を採用できるために宿舎などの福祉にかかる経費を節約できるという利点があります．ただ，電子産業の部品製造のような単純労働

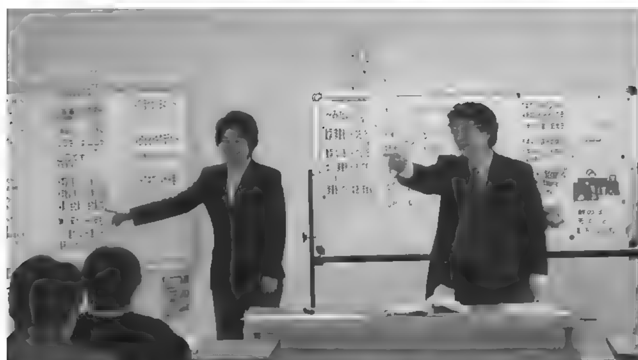


写真４　日本語教室で中国語で挨拶する筆者（右），左は日本語教師の中村紀子さん

は，10年以内にこの街では単価的に成り立たなくなると想定される会社があります．

● 簡単な情報産業

この分野は日本からの発注に依存して経営を成り立たせています．ただ，製造業ほど単純な作業ではなく，多くの場合は日本語ができる労働者を必要とします．

合肥市の日本語教育環境は，あまり良いとはいえません．日本語教師が不足しているだけでなく，教育施設や教師の質といったこと以外に，彼らに日本語を学んで仕事の役に立てようという意識があまり大きく芽生えていないのも原因かもしれません（写真４）．

これは進出，あるいは外注を行う企業の側で独自に考えるか，地方政府に要望を出すなどの必要があります．日本語ができる人材が先か，日本語の仕事が先という議論が発生しますが，基本的には発注側がどのくらいの費用と時間をかけられるかにかかっています．この点ではあとで触れる大連市など日本語の土壌がある都市とは大きく事情が異なります．

● IC設計やシステム開発業

この分野は中国側が一番期待している分野です．しかし，新人の中国人技術者は本人が考えているほどすぐには役に立つとはいえず，現時点では外注での開発依頼は，日本や欧米で訓練を受けた技術者がいない企業には行わないほうがよいといえます．

たしかに中国人の有力大学・大学院卒業生はたいへん頭が良く，知識も豊富で発言も積極的で英語の能力も十分といえます．しかし，経験面からいうと2003年の中国全土のIC設計関係の従業員総数7,000名の中で60%が経験３年以下という状況です．

優秀な中国人技術者の多くは，自己主張が強く日本的な開発組織に適合しにくいことがあります．欧米人も同じ傾向がありますが，それは開発の方式をシステム化することで解決しています．一般の中国人は，人間関係が強く出る傾向があるので，日本的な組織や欧米式のシステムを受け入れるには，ある程度の研修が必要です．

さらに，中国企業も独自開発を行うには資金の点で不安があり，請け負いのように全面的に責任を持つ仕事では，企業が立ち行かなくなった場合の保証が弱いという現実があります．中国人の考え方として保守というものをあまり重要視していないことも原因かもしれません．

現在，合肥市にハイテク産業の企業として進出するには，日本で数年の研修を済ませた技術陣が1/3程度は必要です．そのためには，日本での技術者の教育・訓練投資が必要となります．安易に中国で開発すれば安くあがると考えるのは軽率ですが，本気で進出を考える企業には，コストと人材の面で有利な面が多々あると思われます．

● 日本からの就職先

日本人の就職先として合肥市は現時点ではほとんど期待できません．外国人技術者がSilicon ValleyなどでIC設計のフリー技術者として渡り歩く姿は，合肥市ではあまり見られないようです．

それは，受け入れ企業の数や給料という壁とともに，中国語の壁が高いからです．日本人の技術者で中国語ができる人がほとんどいない状況では，合肥市が10年後にたとえSilicon Valley並みになったとしても，日本人が職を探すことは難しいかもしれません．大連のように多くの技術者が日本語を話せるようになると，中国なみの給料でよければ就職先が見つかるかもしれません．

やはり，日系企業に就職して現地に派遣されるか，日本語教師の資格を得て現地の技術者に日本語と日系企業での仕事のやり方を教えることなら可能性があります．

また，高齢の技術者の方はJICAなどのボランティア（中国語では「花甲専家」という）で現地費用の保証をもらって行くしかないでしょう．そのような状況下なら，通訳を付けてもらうことは可能です．ただ，通訳は同時通訳などができる専門家としての訓練を経ている人はまれで，日本からの帰国技術者に頼ることになります．

● 大連市の現状

合肥市以外のすでに発展した研究学園都市型の中国のIT産業基地としては，前回にも触れた北京市の中関村や大連市があります．

大連市は清国を立てた女真族（満州族）の根拠地であった瀋陽を省都とする遼寧省の南端に位置する遼東半島の突端にあり，有名な軍港旅順を抱える人口約560万人の産業都市です．農村人口が少なく，日本企業の対中進出の拠点ともなっていて，日

本人も多く駐在しています．大連には駐在者の子弟のために日本人学校があり，瀋陽の日本領事館は，北朝鮮の脱北者が駆け込んだことで一躍有名になりました．

この大連市のIT産業の状況はNHKでも7月に放映されました．それによると，大連市はIT関連産業で外国と中国大陸部との接点になろうとしています．一方，国内向け出荷という点では，陸路では省都の瀋陽市の近くまで渤海湾を大きく迂回して行かねばならないので，目と鼻の先の韓国や日本のほうを向いた産業が栄えています．

日本からの駐在者のためだけではなく，一般向けに日本語のテレビ放送や学校での日本語教育も行われています．中高生の約1/3が日本語教育を受けているそうです．戦前，日本が植民地で無理やり日本語教育を実施したのとは様変わりです．ちなみに，大連市のWebサイトは外国語は英語だけでなく図3のように日本語やハングルでも閲覧できます．

さらに，中国全土では現在情報系の学卒者が毎年18万人いると言われており，日本の2万人程度と比べるとべらぼうというしかない人数で，さらに増えつつあるようです．それらの学生にどのように教育をしているかが報告されていました．

大連市には東北大学東軟信息技術学院という情報技術専門の大学があり，全校で六千人の学生が全員パソコン持参で授業を受けているとのことです．この大学は瀋陽市にある情報科学・技術では中国有数といわれる東北大学の関連企業の東軟集団が，自社の要員を効率的に育てるために作ったものだそうです．

東軟集団は，みずからが主導して作った大連市のソフトウェア・パークに基盤を置く，中国最大の独立系IT企業で，多くの日本企業からソフトウェア開発の受注をすることで売り上げを伸ばしてきました．大連市にはこれら中国の会社のほかに日本から進出した企業も多く，東芝，日電，ソニーなどがソフトウェア・パークに拠点を作っています．日本の大手会社が政府からの受注をここへ丸投げしていた問題も発生しています．

ソフトウェア・パークに進出した60社を越える外国企業の内2/3は日本からの進出で，多くは東軟集団との合弁企業です．それらの企業は事務処理や日本語処理のソフトウェアのような下請け作業を主に行ってきました．

最近は簡単な作業だけではなく，このような優秀な人材の供給能力を背景に，現在日本では良質な人材確保が難しい組み込みソフトウェアの分野に進出しつつあるようです．日本のあるカーナビやカー・オーディオの会社では，ソフトウェアの設計業務の90%をすでに大連市で行っているそうです．

さらに，週刊誌の記事によれば，日本で仕事にありつけなかった日本人も，大連市の日系企業で現地採用者として時給20元（月200時間働くと約5万円）程度で，日本語処理や打ち込みを主体とした仕事に就いているという話です．

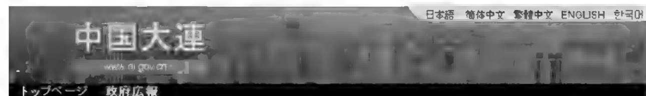


図3　大連市の五種の（繁簡2字体を含む）文字によるWebページ（http://www.dl.gov.cn/i18n/jp/）

● 珠江デルタ地区ではどうか

中国には研究学園都市ではなくてもIT関係の産業が集積している都市があります．その一つ上海市には，理工系で有名な上海交通大学があります．広州市には辛亥革命の父孫文の号を取った中山大学がありますが，この大学の理系は医学などで有名です．どちらの地域も大学からの発信を主体とした産業基地ではありません．

香港の対岸にある広東省の省都広州市といくつかの市を含む珠江デルタ地帯は，人口3,000万人を有する，1979年の改革開放政策の開始後最初に発展した地域です．ここにはIT産業のみならず，食品・自動車などいろいろな産業が世界中から投資しています．台湾や日本からの進出企業数も計5万社を越えています．

珠江デルタ地区は世界最大の工場地帯となりつつあり，上海を中心とする長江下流域（江南）地帯とともに，輸出を主体に発展を続ける中国経済の牽引車となっています．

珠江デルタでのIT産業の特徴は，早期に発展したこともあり，コピーやプリンタ，パソコンとその部品などを中心としたハードウェア製造の企業が集中していることです．世界のコピー・プリンタの半分，アジアに出まわるパソコン本体のほとんどはこの地域で作られていると考えてもよいでしょう．

その意味ではこの地域にはあらゆる産業がありますが，日本人のIT技術者が単独で職を求めるとなると，英語や中国語に堪能でない限り日系企業を探すしかないでしょう．街の人の日常言語は広東語が圧倒的に強いことと，食生活が大きく違うので，中国人と混じって住むには適していないかもしれません．

● おわりに

先月号では，世界のハイテク産業の集積地をざっと見渡しましたが，幸い中国の合肥市からその状況を日本に知らせるべく現地への視察を要請され，見学して来たので紹介しました．技術者はこれからの仕事の場所を全世界に求めていく必要があります．中国への進出や外注を考えている企業にも，参考になることがあれば幸いです．

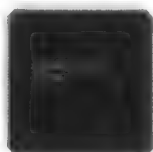
いかい・くにお　エム・アイ・ベンチャー（株）

●32ビットRISCマイコン

SH7206

• 2命令同時実行可能なアーキテクチャのスーパ・スカラ方式を採用し，リアルタイム制御性能を向上した新規開発のCPUコア「SH-2A」(360MIPS/200MHz動作時)を採用.
• CPU内部に割り込み専用のレジスタ・バンクを15面搭載し，割り込み処理までの応答時間を短縮.
• 1サイクルでのアクセスが可能な128Kバイトの大容量RAMおよび16Kバイトのキャッシュ・メモリを内蔵.
• ACモータなどの制御が可能な多機能タイマ・ユニットを2系統搭載しているため，二つのモータを同時に制御することが可能.
• 各種産業機器向けに有用な10ビットのA-Dコンバータ，8ビットのD-Aコンバータを搭載.

●サンプル価格：¥2,100



■（株）ルネサス テクノロジ
TEL：03-5201-5214
E-mail：csc@renesas.com

●ニューラル・プロセッサ

SILIMANN 120CX

• 人間の脳のしくみに近いアナログのニューラル・ネットワークで，数μsの処理が可能.
• 教育することで，人間に近い感覚で高速演算を行う.
• 信号とパターンを教育することで，プログラミングを行うことなく稼動させることが可能.
• アナログ信号を直接入力することができ，高速処理に加えてA-D変換回路を省くことが可能.
• ニューラル・ネットワークによって解を見つけることが困難な数学的モデルの代わりに，教育データを使用することで非線形適応システムを実現.
• ロボットの姿勢制御，あいまいな色の判断，自動車事故軽減など，データとパターン識別が必要とされる市場に適する.

●価格：下記へ問い合わせ

■ユニイット（株）
TEL：03-5251-8101 FAX：03-5251-8120

●8ビット・マイコン

M37544G2A

• 新規開発のプログラマブル・メモリ「QzROM」を内蔵したことで，ROMコードを受注して出荷するまでの期間を，約1/2以下に短縮.
• 「QzROM」は，マスクROM版相当の価格を実現し，不正な読み出しを不可能にするプロテクト機能を搭載することで，耐タンパ性を向上しており，高セキュリティを実現.
• 動作電圧範囲を，従来の4.0～5.5Vから，1.8～5.5Vにして低電圧側を拡大.
• プロセスは0.35μm CMOSを採用し，最大動作周波数は8MHz.
• 8KバイトのROM，256バイトのRAMを搭載.

●サンプル価格：¥200（M37544G2ASP）
¥184（M37544G2AGP）



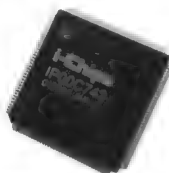
■（株）ルネサス テクノロジ
TEL：03-5201-5037
E-mail：csc@renesas.com

●カラー画像用解像度変換LSI

IP00C743(SCL3)

• 液晶ディスプレイなどのドット・マトリクス型の表示デバイスに必要不可欠なカラー・ディジタル画像の拡大を1チップで実行するLSI.
• NTSCをはじめとしてSXGAまでの広範囲な画像入力に対応でき，外付けのフレーム・メモリなしで，高速，高品位な画像拡大処理を実現.
• ラスタ・スキャンのディジタル画像入出力と4線シリアルのCPUインターフェースを接続するだけで，画像の拡大処理システムの構築が可能.
• 水平/垂直4シンボル・プログラマブルFIRフィルタを搭載することで，高画質の拡大表示を実現.
• 縦横独立な倍率設定が可能.

●価格：下記へ問い合わせ



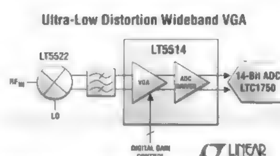
■アイチップス・テクノロジー（株）
TEL：06-6492-7277 FAX：06-6492-7388

●高出力レベル広帯域PGアンプ

LT5514

• 70MHzで49dBmのOIP3（Output third-order intercept）を提供し，2次または3次歪み積が－87dBc以下で，200Ωの負荷に対して2Vピーク・ツー・ピークの出力を供給.
• 低ノイズでスプリアスのない高出力を供給できるので，高性能通信システムにおいて12または14ビット，65Msps以上のA-Dコンバータのドライバとして使用可能.
• 利得は，内蔵のディジタル制御減衰器と固定利得アンプによって可変に調整される.
• 4ビットのパラレル入力ワードによって，10.5～33dBの範囲にわたり1.5dBの増加ステップで利得の制御が可能.
• 差動入出力方式を採用し，低歪み特性を保持しながらA-Dコンバータを平衡ドライブ.

●サンプル価格：¥601（1,000個時）



■リニアテクノロジー（株）
TEL：03-5226-7291 FAX：03-5226-0268

●ステレオ・アンプ・ファミリ

WM8602

• 2チャネルのディジタル音声を入力とし，完全オーディオ帯域幅出力を2チャネルとLFEサブチャネル1チャネルのPWM出力が可能.
• 各メーカから供給されるスイッチング・パワー・ステージ，または任意のプリ・ドライバおよびFETとともに，さまざまな出力アンプに使用可能.
• ディジタル・クリッピングを防ぐダイナミック・ピーク・アーキテクチャや，トーン・コントロール，バス・コントロールなどの再生音質向上機能を内蔵.
• バス・コントロール，ディジタル信号処理とフィルタリング，およびクラスD出力段を駆動可能なPCM/PWM変換に加え，オンボード・グラフィック・イコライザと各種の高周波数イコライジング機能があり，さまざまなタイプのスピーカに適合.

●価格：下記へ問い合わせ

■ウォルフソン・マイクロエレクトロニクス・ピーエルシー
TEL：03-5733-8677 FAX：03-3578-1488

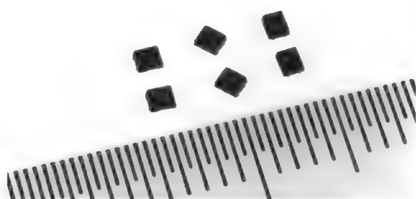
■価格は税込みです.

●ボルテージ・ディテクタ

S-1000 シリーズ

- 携帯機器用電源の監視，定電圧電源の監視，マイコン用電源の監視および CPU リセット，バッテリ・チェッカ，停電検出器などの用途に適する．
- 微細プロセス技術の開発や回路設計の最適化などを行い，従来品と同等の応答速度を確保し，350nA（typ.）の低消費電流を実現．
- ±1.0%の高精度検出電圧により，電池（電源）電圧を高精度で検出可能．
- 電源検出回路の設計が容易となる．
- 精密加工技術をもとにした，新開発の小型 SNT-4A を採用し，小型化（1.20×1.57×0.48mm）を実現．

●サンプル価格：¥105



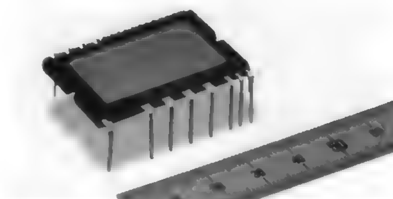
■セイコーインスツルメンツ（株）
TEL：043-211-1193

●インバータ駆動用パワー半導体モジュール

DIP-IPM Ver.4 シリーズ

- 放熱性に優れた新絶縁構造を採用することにより，熱抵抗を低減し，パワー半導体モジュールの温度上昇を抑制．
- 従来品と比較して，実装面積（38×24mm）を約 60%縮小．
- パワー素子の鉛フリーはんだ付けプロセスの導入により，高信頼性を確保しつつ内部の鉛フリー化に対応．
- 出力素子耐圧は 600V で，コレクタ電流は 5/10/15A．
- 三相インバータを構成する IGBT チップ，FWD チップ，HVIC チップ，LVIC チップを内蔵．
- 制御ユニットからの直接接続が可能．
- 単電源での駆動が可能．

●サンプル価格：¥1,418～¥2,100



■三菱電機（株）
TEL：03-3218-4818 FAX：03-3218-2723

●出力 OP アンプ

ADA4851-x ファミリ

- ADA4851-1 はシングル，ADA4851-2 はデュアル，ADA4851-4 はクワッドの電圧帰還式レール・ツー・レール出力 OP アンプ．
- 175MHz の帯域幅，250V/μs のスルーレート，および 25ns で 0.1%のセトリング時間と 1アンプあたり 3mA の静止電流を実現．
- 単一電源動作で，入力での信号レベルは負電源以下 200mV まで拡張して使用可能．
- 出力は両電源電位から 50mV 以内の電圧までスイング可能．
- ゲイン平坦性は帯域幅の外 15MHz まで 0.1dB で，ゲインおよび位相の変動はそれぞれ 0.05%および 0.05°となっており，ビデオ・アプリケーションに適する．
- 標準品位ビデオ，通信，自動車テレマティクス向けに，高品質の信号処理を実現．

●サンプル価格：
ADA4851-1 $0.39（100,000 個時）
ADA4851-2 $0.55（100,000 個時）
ADA4851-4 $0.79（100,000 個時）

■アナログ・デバイセズ（株）
TEL：03-5402-8125

●スマート・カード向けセキュア IC

ST19WR66

- 8ビット MCU と周辺機能を内蔵したスマート・カード向け IC．
- 66K バイトの EEPROM を搭載しているため，現行の ICAO（国際民間航空機関）プログラムで要求される，バイオメトリクス・データと個人情報の格納が可能．
- 224K バイトのユーザ ROM は，OS とプログラム・コードの格納が可能．
- 6K バイトのユーザ RAM は，8ビット・セキュア MCU の処理性能とあわせて，発行段階での迅速なパスポート/ID のパーソナリゼーションや現場での即座の照合に不可欠な高速データ処理を可能にする．
- 公開鍵暗号方式の 1088 ビット MAP（モジュラ演算プロセッサ），強化版ハードウェア DES エンジン，および AES-128 ソフトウェア・ライブラリ機能を装備．
- 機密性の高い個人データ・レコードの格納，認証およびディジタル署名の提供に必要なデータの暗号化/復号化機能を備えたアプリケーションを提供．

●価格：下記へ問い合わせ

■ST マイクロエレクトロニクス（株）
TEL：03-5783-8320 FAX：03-5783-8216

●ディジタル・アンプ・コントローラ

CS44600/CS44800

- 特許技術であるパワーサプライ・リジェクション機能により，シングル・エンド出力で音質に影響する電源ノイズの自動補正が可能．
- 音質向上により，低価格な電源の使用が可能．
- チップに集積化されたサンプル・レート・コンバータを搭載することで，クロック・ジッタの影響を最小限にし，システムにおけるダイナミック・レンジ 100dB 以上の高音質を実現．
- ポップガード技術が組み込まれており，シングル・エンド出力に頻繁に発生するクリックとクロック・ノイズを最小限に抑える．
- 7.1チャネル・リファレンス・デザインに対応しており，迅速な市場投入が可能．
- オーディオ/ビデオ・レシーバや DVD レシーバ，ホーム・シアタ・レコーダなどの次世代ホーム・シアタ製品向けの製品．

●サンプル価格：CS44600 $3.27（10,000 個）
CS44800 $3.55（10,000 個）

■シーラス・ロジック（株）
TEL：03-3261-7715

●電源制御 IC

Powerlinker

- 薄型テレビなどで必要とされる多チャネル電源の起動，静止制御が可能な DC-DC コンバータ．
- システム異常で電源停止する際も，複数の電源間での停止の連動が設定可能．
- 多数の DC-DC コンバータ出力が必要な場合，製品を複数使用することにより，外付け部品の追加なしに IC 間の連携を実現可能．
- 入力電圧自体の異常を検出する，System_UVLO 機能を搭載．
- 検出電圧は，外付け部品の選択により任意に設定でき，システムごとのニーズに対応した設計が可能．
- 位相をずらしたスイッチング方式の採用により，入力側に回り込むノイズを削減可能．
- 負荷電流に依存しないソフト・スタート時間の設定が可能．

●価格：下記へ問い合わせ

■ザインエレクトロニクス（株）
TEL：03-3270-0666

■弊誌では新製品に関するニュースリリースを募集しております．
宛先は，〒170-8461 東京都豊島区巣鴨 1-14-2 Interface 編集部ニュースリリース係
FAX：(03)5395-2127，E-mail：mngnews@cqpub.co.jp （編集部）

●ハロゲン・ランプ・コントロール評価ボード

IRPLHALO1E

- ハロゲン・ランプ用インテリジェント・コントロールIC「IR2161」の評価ボード.
- 設計工程の簡略化，製品化までの期間短縮を実現.
- 回路は，ハロゲン・ランプ用に最適化して設計されている.
- 電子変圧器の標準回路は，補正済みの負荷レギュレーションと，過熱シャット・ダウンによる短絡，過負荷防止機能を備え，あらゆる種類の調光装置と互換性を持つ.
- EMI（電磁干渉）フィルタを内蔵.
- 回路図，部品表，基板配置図，アプリケーション・ノート，基板シルク・スクリーン，データ・シートなどの包括的な文書をサポート.
- バラスト（安定器）の設計ソフトウェアを含む.

●価格：下記へ問い合わせ

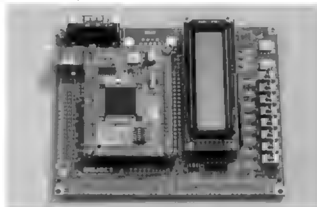
■インターナショナル・レクティファイアー・ジャパン（株）
TEL：03-3983-0086 FAX：03-3983-0642

●オンボード・デバッグ・ユニット

AM1 Debug Probe Dwire

- 松下電器産業社製マイコン「AM1シリーズ」対応のデバッグ・ユニット.
- サポート品種は，MN101CF73A/74Gの2品種.
- デバッグ・モニタの書き込みが不要で，強制ブレーク，実行前ブレーク，イベント・ブレーク，ステップ実行，メモリおよびレジスタの参照や変更などが行える.
- 付属するデバッガは，AM1マイコンのうち1品種をサポートした統合開発環境の「Debug Factory Builder」.
- エディタ，メイク，デバッガの切り替えができ，Cソース・コード・レベルでのステップ実行，インスペクト，ウォッチ，ブレーク・ポイント設定などが行える.

●価格：¥73,290



■（株）オブジェクト
TEL：06-6844-1747 FAX：06-6844-1760
E-mail：info@object.co.jp

●メジャリング・レシーバ・システム

N5530S メジャリングレシーバシステム

- 最大26.5MHzの周波数帯までダウン・コンバータなしで，周波数のカウント，絶対パワー測定，同調RFレベル測定，変調解析（AM，FM，PM）測定，変調歪み測定，スペクトラム解析など，信号発生器やアッテネータの校正に必要な機能を提供.
- スペクトラム・アナライザ「PSAシリーズ」，パワーメータ「EPMシリーズ」をベースに，新開発のセンサ・モジュールと専用PCソフトウェアを組み合わせたシステム.
- 従来品と比較して，信号のロスを4dB低減.
- 被測定物からの信号の切り替えに，パッシブ・スプリッタを用いることで，測定の再現性，信頼性，性能の向上を実現.

●価格：約¥8,000,000～



■アジレント・テクノロジー（株）
TEL：0120-421-345

●ディジタル波形発生器/アナライザ

NI PXI-6542, NI PXI-6541

- 「NI PXI-6542」では最大100MHzのクロック・レートを,「NI PXI-6541」では最大50MHzのクロック・レートを実現.
- チャネルごとに入出力の方向制御が可能な32チャネルを備える.
- 5.0V，3.3V，2.5V，1.8Vなど，広範な共通ロジック製品のインターフェースとして利用できるよう，電圧のプログラミングが可能.
- SMC（Synchronization and Memory Core）アーキテクチャを採用することで，複数のディジタル・モジュールの同期多チャネル・アプリケーション，およびディジタル・モジュールとアナログ・モジュールの同期ミックスド・シグナル・アプリケーションの構築が可能.

●価格：¥690,000（NI PXI-6542）
¥414,000（NI PXI-6541）



■日本ナショナルインスツルメンツ（株）
TEL：0120-527196 FAX：03-5472-2977
E-mail：salesjapan@ni.com

●産業コントローラ・モジュール

SCM220, スターターキット

- CAN，FPGA，LANネットワーク（10Base-T），RS-232-C，RS-485，スマート・メディア・ディスク（16M相当）を標準で装備した産業用コントローラ・モジュール.
- 拡張モジュールにより，I/O入出力，A-D/D-Aコンバータ，Profibus，モーション・コントローラ，ビデオ入出力などに対応.
- DC12～24Vで駆動し，DINレールに装着可能.
- ソフトとしてISaGRAFを利用でき，ラダー言語などでモジュールをコントロール可能.
- ITRONに準拠したオープン・ソースOSを付属し，自由なプログラミングが可能.
- Eclipseをベースとした，統合開発環境をサポート.

●価格：下記へ問い合わせ

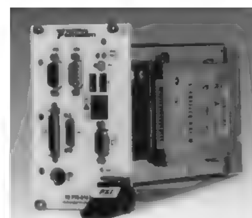
■（有）シンビー
TEL：043-244-9714

●PXI組み込みコントローラ

NI PXI-8187

- Pentium4-M（2.5GHz），および最大1Gバイトの DDR-SDRAMを搭載.
- ハードディスク・ドライブ，シリアル，USB，GP-IB，Ethernetなどの周辺装置を，単一のモジュールに組み入れることができるため，システムの統合が容易.
- すべてのPXIおよびCompactPCI周辺モジュールに対応しており，OSはWindows XP/2000，またはNI LabVIEW Real-Timeのいずれかを選択可能.
- 日本語，英語，スペイン語，ドイツ語，韓国語，フランス語，中国語版Windowsに対応.

●価格：¥647,000



■日本ナショナルインスツルメンツ（株）
TEL：0120-527196 FAX：03-5472-2977
E-mail：salesjapan@ni.com

■価格は税込みです.

●USB対応データ集録製品

NI DAQPad-6015, NI DAQPad-6016, NI USB-9211, NI USB-9215, NI SCXI-1600

- 「DAQPad-6015」,「DAQPad-6016」は，16ビットの確度を備え，1チャネル使用時に最大200kspsのサンプリング・レートを実現したデータ集録モジュール．ねじ式端子台を搭載．
- 「USB-9211」は，ねじ式端子台を搭載した4チャネル，24ビットの高確度熱電対計測デバイス．「USB-9215」は，位相遅延が少なく，複数の信号を高い確度で計測できる4チャネル，16ビットの同時サンプリング・アナログ入力デバイス．データ・ロギング・ソフトウェアが付属．
- 「SCXI-1600」は，フル機能搭載の16ビットUSBデータ集録/制御用モジュール．NI SCXI内の40の計測モジュールに対して，プラグ&プレイで接続可能．SCXI信号調整ハードウェアに接続されていたDAQデバイスは不要となり，1本のケーブルをコンピュータに接続するだけでシステムの構成が可能．

●価格：¥165,000（NI DAQPad-6016）¥138,000（NI DAQPad-6015）/¥62,000（NI USB-9215）/¥62,000（NI USB-9211）/¥138,000（NI SCXI-1600）

■日本ナショナルインスツルメンツ（株）
TEL：0120-527196 FAX：03-5472-2977
E-mail：salesjapan@ni.com

●パラレル-USB変換器

USB-006

- パソコンやプリンタ・インターフェースを持つ機器からの印刷用出力データを，パソコンに取り込むことが可能．
- オシロスコープや測定器からのHPGL出力をファイルに変換し，活用することが可能．
- Windows上で仮想COMポート・ドライバの機能により，COMポートとして動作するので，外部から入力されたデータを取り込むことが可能．
- パラレル側は，付属ケーブルで標準的な25ピン・コネクタに接続可能．
- 対応OSは，Windows 98/Me/2000/XP．
- 使用しているチップ・メーカからドライバをダウンロードすれば，MacやLinuxでも動作可能．

●価格：下記へ問い合わせ



■（有）ヒューマンデータ
TEL：072-620-2002 FAX：072-620-2003

●DVD/HDカセット・プレーヤ

Mpeg HDGate

- テレビ側に設置して使用する据え置き型のDVD/HDDプレーヤ．
- DVDビデオや記録型DVDドライブで書き込まれた，マルチメディア・データ・ファイルの再生が可能．
- 独自システムである「HDゲート」方式の採用により，パソコンのHDDに保存されている映像，音楽，写真データなどを，テレビやプロジェクタで再生可能．
- HDDをセットしたリムーバブル・ハードディスク・ケース「HDカセット（ハーディーカセット）」をMpeg HDGate本体にセットしてコンテンツを再生できる．
- 再生可能なファイル形式は，DVD-Video，DVD-VR，SVCD，VCD，MPEG-1/2/4，DivXなどの動画ファイル，JPEG静止画像ファイル，音楽CD，MP3などの音声ファイル．
- 再生可能なメディアは，DVD-R/RW，DVD+R/RW，DVD-ROM，CD-R/RW，CD-ROM．

●価格：オープン

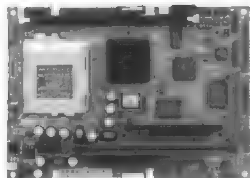
■（株）ノバック
TEL：03-3817-7826

●小型CPUボード

SLC-8150-LVA

- Intel Pentium III（500MHz～1.26GHz），Celeron（300MHz～850MHz）CPUを搭載可能な，5インチ・ベイサイズのシングルボード・コンピュータ．
- チップ・セットにVGAコントローラを内蔵する，Intel 815E/ICH2を採用．
- 最大512MバイトのSDRAMメモリを実装可能であり，シリアル×4，USB×4，100Base-TX LAN×1，パラレル×1，EIDE×2，Disk On Chipソケット，AC97オーディオなどのインターフェースを標準装備．
- PISA，PC/104，MiniPCIの拡張バス・スロットを装備しており，機能拡張が可能．

●価格：¥59,850



■（株）コンテック
TEL：03-5628-9286 FAX：03-5628-9344
E-mail：tsc@contec.co.jp

●ビル間通信用無線LANユニット

SB-510/SB-510EA

- 「SB-510」は，省スペース設置が可能なアンテナ内蔵タイプ．
- 「SB-510EA」は，アンテナ外付けタイプで，最大伝送距離約4km（指向性ロング・アンテナ同士）を実現．
- オプション・アンテナは，指向性ロング/指向性/無指向性/平面など6種類のアンテナ・タイプを用意．
- 離れた建物間を，IEEE802.11g準拠の高速無線LAN（Super G対応）で接続．
- 本体の設定を変更することにより，IEEE 802.11b（2.4GHz，11Mbps）通信が可能．
- 高度な暗号化セキュリティOCB AESを搭載．
- 普及タイプのWEPでは，64/128/512ビットの暗号化伝送方式を実現．
- PoE対応（802.3af準拠）により，LANケーブルで電源供給が可能．

●価格：¥129,150

■アイコム（株）
TEL：06-6792-4949

●CFメモリ・カード制御基板

CF-Board.ing

- ユーザが用意したマイコン基板と本製品をRS-232-Cで接続することにより，CFメモリ・カードへの読み書きができる．
- FAT12/16/32（今秋予定）に対応．
- 同期式通信の場合，通信速度は最大921.6 kbps．
- RS-232-Cまたは，3線式EEPROM制御方式に近いI/Oで制御を行い，BUSY信号は省略可能，最大4本の制御線で接続が可能．
- 調歩同期式通信の場合，通信速度は一般的な9600bps系列で，最大460.8kbps．
- BUSY信号でCFメモリ・カードの状態を監視することができ，マイコン基板の割り込み入力に接続することで，制御効率が向上．通信バッファ制御信号としても利用が可能．
- パソコンでカードの内容を確認でき，CFメモリ・カード利用装置の検証に利用できる．
- 独自フォーマットによる直接制御も可能で，データの機密性を向上．

●価格：下記へ問い合わせ

■（株）高崎共同計算センター
TEL：027-326-8160 FAX：027-326-8904
E-mail：can2@ms.tkcc.com

■弊誌では新製品に関するニュースリリースを募集しております．
宛先は，〒170-8461 東京都豊島区巣鴨1-14-2 Interface編集部ニュースリリース係
FAX：（03）5395-2127，E-mail：mngnews@cqpub.co.jp （編集部）

●無停電電源装置

BU50XS

- 常時インバータ給電方式のため，出力電圧はつねに一定しており，停電発生時にも切り替え時間の発生しない無瞬断出力.
- UL規格を取得しており，本体，バッテリを含めた3年間保証付き.
- 期待寿命4～5年の長寿命バッテリを搭載しており，フロント・パネル側から交換可能.
- 標準添付のネットワーク対応自動シャットダウン・ソフト「PA」は，Windows Server 2003およびLinuxに対応.
- 「PA」を使用することにより，複数のサーバ/パソコンの連携シャットダウン，スケジュール運転などが可能.
- Visual Basic，Visual C++のサンプル・プログラムをサポートする，UPS制御用ライブラリ・ソフト「UPSライブラリ」は無償でダウンロード可能.

●価格：¥88,200

■オムロン（株）
TEL：03-3436-7228

●マルチメディア・プラットホーム

CSB-50

- CPUに，Intel XScale IXP425（533MHz）/IXP420（266MHz）を搭載した組み込み向けマルチメディア・プラットホーム.
- NTSCカメラを4台接続可能なカメラ入力端子（RCAピン・ジャック）を装備.
- 高速画像圧縮伸張機能「Dual RAPIC」を持つ，アクセル社のグラフィックLSI「AG902」を搭載.
- データ保存用メディアとして，Compact Flash用カード・スロットを装備.
- コンソール用VGA出力やタッチパネル用シリアル・インターフェースのほか，USB，Ethernet（10Base-T/100Base-TX），アイソレーションIN/OUTを搭載.
- ボード上に拡張ボードを搭載可能.
- DC+12V（2.5A）の単電源駆動.
- 新世代の組み込みプラットホームとして，高機能，省電力で，広範囲なマルチメディア機器に対応.

●予定価格：¥168,000

■（株）アバールデータ
TEL：042-732-1030

●デバイス・ドライバ開発ツール

WinDriver Windows CE v6.22

- Jungo社のデバイス・ドライバ開発ツール.
- Windows CE 5.0に対応.
- PCIおよびUSB1.1/2.0のWindows CEデバイス・ドライバをユーザ・モードで開発可能なツール・キット.
- OSの内部構造，カーネル・レベルのプログラミングの知識は不要.
- Windows 98/Me/NT/2000/XP/Server 2003，Windows CE.NET，Linux，SolarisおよびVxWorksに対応し，開発したコードはOS間での互換性を維持.
- ウィザードによるグラフィカルな開発環境，API，ハードウェア診断ユーティリティおよびサンプル・コードを提供.
- ドライバ・コードの自動生成，およびドライバのデバッグ環境をサポート.
- 主要なチップ・ベンダに対する拡張をサポート.

●価格：¥386,400（WinDriver CE）
¥516,600（WinDriver USB for Windows CE.NET）

■エクセルソフト（株）
TEL：03-5440-7875 FAX：03-5440-7876
E-mail：xlsoftkk@xlsoft.com

●リアルタイム・モジュール

LabVIEW 7.1 Real-Timeモジュール

- 「LabVIEW」で開発したプログラムを，RTOS上で実行させるためのアドオン・ツール.
- 「NI-DAQmx」への対応により，外部I/Oを含んだシングル・ループ・PIDアプリケーションの処理速度を30%向上.
- ハードウェア・タイミングのループを，より高い確定性をもって実行可能.
- 複数のI/Oボードの同期，確定的なフィードバック処理などの機能を短時間で実装可能.
- リアルタイム・システム実行中のCPUやメモリの使用率を表示することができるため，高度なデバッグを実現.
- PCIボード，PXIシステム，FieldPoint，Compact Vision Systemなどのハードウェアに加えて，特定のデスクトップPC上でもリアルタイム・システムの構築が可能.

●価格：¥287,000

■日本ナショナルインスツルメンツ（株）
TEL：0120-527196 FAX：03-5472-2977
E-mail：salesjapan@ni.com

●アプリケーション開発ツール

PowerBuilder 10.0

- J2EEと.NET対応のRAD環境.
- ユニコードのサポートにより，国際化対応のアプリケーション開発において，DataWindowの1行内にマルチバイト文字を表示可能.
- 再設計されたXML Web DataWindowにより，パフォーマンス，スケーラビリティ，拡張性が向上.
- PowerBuilder ADO.NETインターフェースの使用によって，Microsoft .NETデータへのアクセス，複雑なデータベース操作の単純化が可能.
- Microsoft Active Accessibilityインターフェースにより，障害を持つエンド・ユーザの使用を支援するアプリケーション開発をシンプル化.
- エンタープライズ・モデリング・ツール「PowerDesigner」用プラグインを使用して，既存のアプリケーションの拡張や異なるアーキテクチャに移行するためのリバース・エンジニアリングが可能.

●価格：下記へ問い合わせ

■サイベース（株）
E-mail：sales@sybase.co.jp
URL：http://www.sybase.co.jp/

●セキュリティ・サービス

SHILDIAN NETservice Ver.3

- ASP方式のセキュリティ・サービスとしては先進的な，ISP機能を実装することで，各種ワームによる攻撃，ポート・スキャンやDoS攻撃，情報漏洩などの不正アクセスを自動的に検知して，遮断することが可能.
- ネットワーク・レベルでの通信監視と，ISPによる総合的な不正アクセス検知/遮断機能により，安心してネットワークへ接続することが可能.
- マルチ・ネットワーク環境での利用を想定した機能を追加.
- IPアドレスなどをキーとして，あらかじめネットワーク環境に応じたファイアウォール・ポリシを関連付けておくことで，それぞれの環境に合ったポリシを自動的に切り替えて適用することが可能.
- LAN上の共有フォルダのアクセスについても，ファイアウォール・ポリシに沿った監視，制御が可能.

●価格：下記へ問い合わせ

■（株）インターコム
TEL：03-3839-6307
E-mail：s-shildian@intercom.co.jp

■価格は税込みです.

●プロジェクト管理ツール

アリエル・プロジェクトA企業版

- 遠隔地のメンバを含んだプロジェクトを円滑に運営，管理するためのソフトウェアに，企業のIT部門で使用するための認証，管理ソフトウェアを同梱したソリューション．
- 分散型のソフトウェアに対してのライセンス付与および無効化，使用権限の変更，使用期限の設定などの認証関連機能をサポート．
- 使用できるネットワーク範囲の指定，利用するネットワークの帯域制御などのネットワーク管理機能のほか，LDAP連携やIPv6混在環境対応，自動データ・バックアップ，復旧などの機能を搭載．
- プロジェクト・マネージャは，セキュリティを確保しながら，社内外のプロジェクト・メンバとタイムリな情報の共有が可能．
- ソフトウェア認証やネットワーク管理を一元化．

●価格：¥3,000,000～

■アリエル・ネットワーク（株）
E-mail：info@ariel-networks.com

●USBストレージ統合キット

GR-USB/HOST，GR-USB/HOSTⅡ，GR-FILE

- 独自開発のUSBスタック「GR-USB/HOST」，「GR-USB/HOST Ⅱ」およびファイル・システム「GR-FILE」を統合パッケージ化．
- ロイヤリティ・フリーで，ANSI Cでのソース・コードを提供．
- μITRON v.2.0/v3.0/v4.0，NORTi，ThreadXなど，各種リアルタイムOSに対応．
- UNIX/Windows，C言語標準I/O互換ライブラリ・インターフェースを提供．
- FAT12/16/32に対応し，ロング・ファイルネーム，Shift-JISファイル名をサポート．
- マルチタスク時の同時アクセスをサポート．
- メディアに応じた，ファイル・システムごとのキャッシング方式の選択が可能．
- 各種コントローラへのポーティング，アプリケーションの受託開発などのサービスを提供．

●価格：下記へ問い合わせ

■（株）グレープシステム
TEL：045-222-3761 FAX：045-222-3759
E-mail：middle@info.grape.co.jp

●分散型システム向けプラットホーム

Enea Orchestra

- 高可用性テレコム/データコム・システム向けに，組み込みLinuxとリアルタイムOSを結合した，統合ソフトウェア・プラットホーム．
- カーネル・デバッグやボードの初期導入から，アプリケーションの開発やテストまで，OSE/Linux製品開発の全工程を簡素化．
- APIをわずかに変更するだけで，Linux用アプリケーションをOSE環境で，またOSE用のアプリケーションをLinux環境で実行することが可能．
- Linux，OSE RTOS，OSEゲート・ウェイ，Polyhedraデータベース，Platform Creation Suite，CodeTest解析ツール，Power Tapデバッグ・プローブ，CodeWarrior開発環境を含むMetrowerks社の開発テクノロジなど，統合された五つのコンポーネントをバンドル．
- Polyhedraデータベースは，組み込みシステム・アプリケーション向けのセキュアかつフォールト・トレラントなデータ・レポジトリを提供．

●価格：下記へ問い合わせ

■エニア・エンベデッド・テクノロジー（株）
TEL：03-5207-6167 FAX：03-5207-6169

●ソフト・エンコーダ

EVC HD MPEG-2 Soft Encoder

- 既存のHD非圧縮編集システムから生成されるファイルをそのまま，ディジタル・ハイビジョンの解像度を持つMP@HLファイル（1080i，720p）に変換可能．
- Bluefish V210 QuickTime CODEC，Avid DS NitrisのAVI出力のファイルの動作検証済み．
- トランスコード・システムとしてのカスタマイズや，オプションでSD解像度対応，HDV対応，MPEG-4（Simple，ASP，Core，Main）対応，H.264/MPEG-4 AVC（Baseline，Main）対応なども可能．
- 指定ディレクトリ中の複数のMPEGファイルすべてにエンコードを行う，バッチ・エンコード・モードでの運用が可能．

●予定価格：¥480,000

■（株）EVC
TEL：03-5687-5841 FAX：03-5687-5843

●リアルタイムLinux

TimeSys Linux

- 通信機器向けにセキュリティ機能を付加したフリースケール・セミコンダクタ社製の通信プロセッサ「MPC8248/8272」プロセッサをサポート．
- IPSec用のセキュリティ・エンジンをサポートし，ソフトウェアIPSecの約5倍（UDPスループット最高66Mbps）の通信速度を実現．
- オープンソースのLinuxカーネルVer2.4をベースとしており，安定したネットワーキングをサポート．
- シングル・カーネル構造と優先度の逆転現象を回避する「プライオリティ・インヘリタンス」機能を持つ．
- オリジナル機能である「リザベーション機能」は，CPU資源やネットワーク資源のコントロールが可能．
- VPNルータ，NAS，IPセット・トップ・ボックス，高機能プリンタ，VoIPゲートウェイなどセキュリティの確保が必要な通信機器，ネットワーク機器の製造分野に適する．

●価格：下記へ問い合わせ

■（株）日新システムズ
TEL：075-344-7961
E-mail：rt-sales@co-nss.co.jp

●工業用部品カタログ

RS総合カタログVol.12

- 電子，電気部品，工具や機構部品などのアイテムを中心に，4,000点の新商品を追加し，約5,000点の商品を掲載．
- ペーパ・カタログとオンライン・カタログ（http://rswww.co.jp/）の形態で展開．
- 基板タイプのナイロン・コネクタや，ねじ式端子台を拡充．
- 鉛フリー対応の電解コンデンサ，ロータリ・コード・スイッチ，カドミウム・フリー接点採用のパワー・リレーを追加．
- ペーパ・カタログでは，10カテゴリで商品群に分け，さらに45セクションに細分化．
- オンライン・カタログでは，Webオーダのほか，多彩な検索機能や約37,000点の商品データ・シートや便利な各種ツールを満載したインフォ・ゾーンを用意．



●価格：
下記へ問い合わせ

■アールエスコンポーネンツ（株）
TEL：045-335-8570 FAX：045-335-8591

■弊誌では新製品に関するニュースリリースを募集しております．
宛先は，〒170-8461 東京都豊島区巣鴨1-14-2 Interface編集部ニュースリリース係
FAX：（03）5395-2127，E-mail：mngnews@cqpub.co.jp （編集部）



INFORMATION
海外・国内イベント/セミナー情報



海外イベント

10/4-8 **PCB Design Conference East**
Expo Center of New Hampshire, Manchester, NH, USA
UP Media Group
http://www.pcbeast.com/

10/5-7 **TECHXNY - NewYork's Technology Week（TECHXNY 2004）**
Jacob K. Javits Convention Center, NewYork, NY, USA
USACMP Media
http://www.techxny.com/

10/12-15 **PTC ASIA 2004**
Shanghai New International Expo Centre, Shanghai, China
Hannover Fairs China
http://www.ptc-asia.com/en/

10/13-14 **Wireless Connectivity Asia 2004（WiCon Asia）**
Shangri-La Hotel Singapore, Singapore
IBC Telecoms & Media Group
http://www.wiconasia.com/

10/13-16 **electronicAsia 2004**
Hong Kong Convention & Exhibition Centre, Hong Kong, China
Hong Kong Trade Development Council
http://www.electronicasia.com/

国内イベント

9/29-30 LinuxWorld C&D/Tokyo 2004
新宿NSビル（東京都新宿区西新宿）
（株）IDGジャパン
http://www.idg.co.jp/expo/lwc/

9/29-10/1 日経ナノテク・ビジネスフェア2004
東京ビッグサイト（東京都江東区有明）
日本経済新聞社
http://www.nikkei-nanofair.com/

10/5-9 CEATEC JAPAN 2004
幕張メッセ（千葉県千葉市美浜区）
CEATEC JAPAN実施協議会
http://www.ceatec.com/

10/13 産総研 知能システム研究部門 研究成果展示会
－オープンハウス2004－
産業技術総合研究所つくばセンター
（茨城県つくば市梅園）
（独）産業技術総合研究所
http://www.is.aist.go.jp/oh2004/

10/13-15 第7回 関西 設計・製造ソリューション展
インテックス大阪（大阪府大阪市住之江区）
リード・エグジビション・ジャパン（株）
http://www.dms-kansai.jp/

10/14-15 組込みソフトウェアシンポジウム2004
日本科学未来館（東京都江東区青海）
情報処理学会ソフトウェア工学研究会
http://honiden-lab.ex.nii.ac.jp/ESS2004/

10/20-22 FPD International 2004
パシフィコ横浜（神奈川県横浜市西区）
日経BP社
http://expo.nikkeibp.co.jp/fpd/index.htm

10/20-23 WPC EXPO 2004
東京ビッグサイト（東京都江東区有明）
日経BP社
http://expo.nikkeibp.co.jp/wpc/

10/27-29 中小企業総合展2004 in KANSAI
インテックス大阪（大阪府大阪市住之江区）
経済産業省中小企業庁
http://www.sougouten.com/

10/28-29 Network Security Forum 2004
青山TEPIA（東京都港区北青山）
（株）IDGジャパン
http://www.idg.co.jp/expo/nsf/

開催日，イベント名，開催地，問い合わせ先の順
日程はすべて予定です．問い合わせ先にご確認のうえ，お出かけください．

セミナー情報

ディジタル変復調の基礎～無線データ伝送からCDMA・UWBまで
開催日時　：10月7日（木）
開催場所　：CQ出版セミナー・ルーム（東京都豊島区巣鴨）
受講料　：13,000円
問い合わせ先：エレクトロニクス・セミナー事務局，☎（03）5395-2125，FAX（03）5395-1255
http://it.cqpub.co.jp/eSeminar/

組み込み型リアルタイムOS（トロン）概要とプログラミング実習
Aコース：組込み型リアルタイムOS概要
開催日時　：10月14日（木）
開催場所　：ポリテクセンター北海道（北海道札幌市西区）
受講料　：10,000円
問い合わせ先：高度ポリテクセンター事業課，☎（043）296-2582，FAX（043）296-2585
http://www.apc.ehdo.go.jp/tron/sapporo.htm

組み込み型リアルタイムOS（トロン）概要とプログラミング実習
Bコース：組込み型リアルタイムOS概要とプログラミング実習
開催日時　：10月14日（木）～15日（金）
開催場所　：ポリテクセンター北海道（北海道札幌市西区）
受講料　：20,000円
問い合わせ先：高度ポリテクセンター事業課，☎（043）296-2582，FAX（043）296-2585
http://www.apc.ehdo.go.jp/tron/sapporo.htm

無線データ通信の基礎と2.4GHz帯無線LAN
開催日時　：10月15日（金）
開催場所　：CQ出版セミナー・ルーム（東京都豊島区巣鴨）
受講料　：13,000円
問い合わせ先：エレクトロニクス・セミナー事務局，☎（03）5395-2125，FAX（03）5395-1255
http://it.cqpub.co.jp/eSeminar/

パソコン実習によるRS-232-CからExcelへのデータ取り込み
開催日時　：10月16日（土）
開催場所　：CQ出版セミナー・ルーム（東京都豊島区巣鴨）
受講料　：25,000円
問い合わせ先：エレクトロニクス・セミナー事務局，☎（03）5395-2125，FAX（03）5395-1255
http://it.cqpub.co.jp/eSeminar/

Verilog-HDL入門
開催日時　：10月21日（木）
開催場所　：CQ出版セミナー・ルーム（東京都豊島区巣鴨）
受講料　：13,000円
問い合わせ先：エレクトロニクス・セミナー事務局，☎（03）5395-2125，FAX（03）5395-1255
http://it.cqpub.co.jp/eSeminar/

USBの基礎と応用
開催日時　：11月4日（木）
開催場所　：CQ出版セミナー・ルーム（東京都豊島区巣鴨）
受講料　：13,000円
問い合わせ先：エレクトロニクス・セミナー事務局，☎（03）5395-2125，FAX（03）5395-1255
http://it.cqpub.co.jp/eSeminar/

TCP/IPによるI/O制御の実際～Ethernetを利用した組み込み機器の設計
開催日時　：11月5日（金）～6日（土）
開催場所　：CQ出版セミナー・ルーム（東京都豊島区巣鴨）
受講料　：25,000円
問い合わせ先：エレクトロニクス・セミナー事務局，☎（03）5395-2125，FAX（03）5395-1255
http://it.cqpub.co.jp/eSeminar/

DSP利用のポイントとプログラミング技術
開催日時　：11月9日（火）～12日（金）
開催場所　：高度ポリテクセンター（千葉県千葉市美浜区）
受講料　：40,000円
問い合わせ先：高度ポリテクセンター事業課，☎（043）296-2582，FAX（043）296-2585
http://www.apc.ehdo.go.jp/

リアルタイムOS基礎コース
開催日時　：11月15日（月）～17日（水）
開催場所　：軽子坂MNビル（東京都新宿区揚場町）
受講料　：31,500円
問い合わせ先：（株）ルネサステクノロジ ルネサス半導体トレーニングセンター，☎（03）3266-9344，FAX（03）3235-5940
http://www.renesas.com/jpn/support/seminar/

ルネサスマイコンセミナー　CANマイコン（M16C/6N）コース
開催日時　：11月16日（火）～18日（木）
開催場所　：アクロス新大阪ビル（大阪府大阪市淀川区）
受講料　：31,500円
問い合わせ先：（株）ルネサステクノロジ ルネサス半導体トレーニングセンター，☎（03）3266-9344，FAX（03）3235-5940
http://www.renesas.com/jpn/support/seminar/

CCDイメージセンサの基礎と応用
開催日時　：11月18日（木）
開催場所　：CQ出版セミナー・ルーム（東京都豊島区巣鴨）
受講料　：12,000円
問い合わせ先：エレクトロニクス・セミナー事務局，☎（03）5395-2125，FAX（03）5395-1255
http://it.cqpub.co.jp/eSeminar/

CMOSイメージセンサの基礎と応用
開催日時　：11月19日（金）
開催場所　：CQ出版セミナー・ルーム（東京都豊島区巣鴨）
受講料　：12,000円
問い合わせ先：エレクトロニクス・セミナー事務局，☎（03）5395-2125，FAX（03）5395-1255
http://it.cqpub.co.jp/eSeminar/

読者の広場

Interface への声



2004年9月号特集「原理から学ぶ ディジタル信号処理技術」に関して

▷以前ディジタル信号処理技術を使いTVのゴースト除去システムを開発していたことがありました．十数年前です．今回の特集はそのときのことを思い出しました．この次は音声や，画像処理技術を取り上げてください．　(ファジィ・ポインタ)

▷CompactPCIに関する詳しい解説が載っている付録が良かった．カード・エッジのPCIとCompactPCIは，電気的にはまったく同じものと思っていたので，REQ64#あたりの信号ピンの処理が異なるなど，勉強になった．　(PCIマニア)

アンケートの結果

興味のあった記事（2004年9月号で実施）

①第1章 ディジタル信号処理技術の歴史
②第3章 DSPで実現する音声処理アプリケーションの開発
③第2章 FIRフィルタの設計と実現方法
④フリーソフトウェア徹底活用講座（第18回）
⑤プロローグ カラーで見る今月の特集
⑥LinuxのMTD（Memory Technology Device）機能を使う
⑦ディジタル信号処理アルゴリズムのハードウェア化手法
⑧ハッカーの常識的見聞録
⑨開発技術者のためのアセンブラ入門（第28回）
⑩第4章 画像処理アプリケーションの作成
⑪第5章 ディジタル変調/復調の基礎と原理
⑫Appendix 実装の心得と勘所
⑬移り気な情報工学
⑭シニアエンジニアの技術草子（四拾弐之段）
⑮MATLAB Release 14の強化機能と追加機能
⑯プログラミングの要（第15回）
⑰Engineering Life in Silicon Valley
⑱RSA Conference 2004 Japan
⑲組み込みプログラミング・ノウハウ入門（第18回）
⑳はじめて使うμClinux（第2回）
㉑C++によるDSPオブジェクト指向プログラミング（第7回）

特集『原理から学ぶ ディジタル信号処理技術』についてのアンケートの結果

Q1　特集全体の満足度をお聞かせください．
①満足した（17％）
②やや満足した（49％）
③どちらとも言えない（17％）
④やや不満である（0％）
⑤不満（17％）

Q2　特集の内容は難しかったですか，やさしかったですか？
①やさしすぎた（17％）
②やさしかった（0％）
③ちょうど良かった（49％）
④やや難しかった（17％）
⑤難しすぎた（17％）

Q3　今後，どのような特集を読みたいたいですか？（具体的にお書きください）
パソコンで試せるディジタル信号処理のサンプル・プログラム集，画像認識技術，USB機器設計，など．

特集担当デスクから

☆エンジニアであれば日常的に何らかの数値と向き合っていることと思います．「測る」，「計る」，「量る」など，何かしらと何らかの量を測定していることでしょう．アナログである一般の世界で起っている現象をディジタルの世界に取り込む際には，必ず「ものを測ること」と，「ディジタル量に変換すること」が必要になります．
☆今回は，ここに焦点を当てました．特に組み込み機器では，すべてがディジタル回路とソフトウェアで収まり切ることは少なく，何かしらのアナログ量と付き合う機会が多いのではないかと考えたからです．
☆最近は，ますますセンサの応用範囲が広まり，機器のユーザ・インターフェースなどが便利になってきています．センサこそ，アナログ量を測るためのデバイスであり，その先には，ほとんどの場合，A-D変換器があるわけです．
☆今回の特集では，そのセンサとA-D変換器を使う前に知っておかなければならないこと――つまりは「測る」ということそのものに注目し，「測る」際に注意しなければならないこと，正しい「測り」方，「測った」あとのデータ処理や蓄積方法，そして「測る」ことによって実現されるアプリケーションの紹介を行いました．

Interface

次号予告

2004年12月号は
10月25日発売です

RFIDのデバイスからシステム構築まで

今，まさに注目を集めているRFID ――無線ICタグに関して，末端のデバイスからデータの管理，バックエンドのサーバ構築までを解説する．

さまざまなところで取り上げられる機会が増えたRFIDだが，その規格および実装系は多くの団体・企業が関わり，さまざまなものが存在する．電波の周波数，デバイスの形状，デバイスの特性…これらに関して単体で解説した記事はあったが，それら全般を俯瞰し，全体の動向を把握した記事はほとんどなかった．そこで本特集では，規格の動向に関して，技術面での分類，そして技術面以外での業界団体の動向について解説する．また，RFIDは電波を扱うことから各種法規制が存在する．現在の規格は法に準拠した形で策定されているが，それらについても解説を行う．

また，RFID技術各論として電磁界シミュレータを用いたRFID機器の設計，バックエンドとなるサーバ構築の実例，市販されているRFID開発キットを用いた開発の実際などを取り上げる．そして，すでに稼動している各規格についても解説を行う．

編集後記

●遺伝子の解明が進むと，「生命の神秘」という言葉が空虚に思えます．IGF・1という成長因子の遺伝子を組み込むと筋肉量が通常より15～30％増え，筋力も倍になるという．脳の発達を促す遺伝子を組み込むと天才が生まれるのでしょうか．アインシュタインの脳や精子はいまも冷凍保存されているようですから…．（檀）

●心理的に気分の悪いことがあると，体調のほうでも気分が悪くなってくる．逆に，体調が良いと気分も晴れるし，調子が悪いと気分も濁ってくる．心と体は一つのものだと実感できる時である．では，いつも気分良く過ごすためには，精神をケアしたほうが良いのか，身体をケアしたほうが良いのか？　鶏と卵だ．（＝IO）

●7割引のエスプレッソ・メーカを新築祝としてプレゼントした．渡した日は引っ越しの直前．その日，その人は雨の中をバイクで埼玉から長野へ急いで帰らなければならなかったので，嫌がらせに近かったかも．引っ越し当日は台風，その1週間後に近くの浅間山が噴火．羨ましくないけれど芳しい人生だなぁ．（もみ）

●携帯の懸賞である公演のペア招待券が当選．新規契約か機種変更が応募条件だったのでわざわざ機種変更した甲斐がある．ちなみにその機種は3万円もしたので連日携帯ショップをはしごして値段を調べた．さらに一番安かったその店のHPからクーポン持参で約半額になった．そしてペア券を当てたので0円．（太陽熱）

●前々から買うと言っていたDVD/HDDレコーダ，やっと（ついに!?）T芝製のを購入．直後に初期不良というトラブルに見舞われましたが，修理後は順調に稼動しております．タイミング的に「オリンピックを録画？」とか言われますが，結局オリンピックは一度も録画せず，くだらない深夜番組などでHDDを埋めています（^^;）（M）

●私のように，持っているCDの95%がインディーズ・レーベルという人にiTunes Music Storeはラインナップの上で不満なのですが，今度始まるrecommuniは，インディーズ・レーベル主催者も関わるという面でかなり期待できそうです．「広く浅く」のiTMSに対し，「狭く深く」の戦略なのかな？　期待しています．（み）

●金16個とその他のメダルを含めて日本は過去最多のメダルラッシュとなったアテネオリンピックも終わりました．日本選手の皆さん，本当にがんばってくれました！　連日，TV中継を見ながら応援して寝不足になった方も多いと思いますが，やはりトップ・アスリート達は人を惹きつける魅力があると思います．（Y2）

●プロジェクトX特別展を観てきました．番組同様，熱い内容でした．中高年の男性が多数を占める観客の中にひとり，展示された電卓のLSIを熱心に見つめる少年を発見．彼は未来のプロジェクト・リーダでしょうか．非常に楽しいイベントでしたが，会場のどこにいても聞こえてくる「あのテーマソング」には閉口…．（と）

お知らせ

■ 読者の広場

本誌に関するご意見・ご希望などを，綴じ込みのハガキでお寄せください．読者の広場への掲載分には粗品を進呈いたします．なお，掲載に際しては表現の一部を変更させていただくことがありますので，あらかじめご了承ください．

■ 投稿歓迎

本誌に投稿をご希望の方は，連絡先（自宅/勤務先）を明記のうえ，テーマ，内容の概要をレポート用紙1～2枚にまとめて「Interface投稿係」までご送付ください．メールでお送りいただいても結構です（送り先はsupportinter@cqpub.co.jpまで）．追って採否をお知らせいたします．なお，採用分には小社規定の原稿料をお支払いいたします．

■ 本誌掲載記事についてのご注意

本誌掲載記事には著作権があり，示されている技術には工業所有権が確立されている場合があります．したがって，個人で利用される場合以外は，所有者の許諾が必要です．また，掲載された回路，技術，プログラムなどを利用して生じたトラブルについては，小社ならびに著作権者は責任を負いかねますので，ご了承ください．

本誌掲載記事をCQ出版（株）の承諾なしに，書籍，雑誌，Webといった媒体の形態を問わず，転載，複写することを禁じます．

■ コピー・サービスのご案内

本誌バックナンバの掲載記事については，在庫（原則として24か月分）のないものに限りコピー・サービスを行っています．コピー体裁は雑誌見開きの，複写機による白黒コピーです．なお，コピーの発送には多少時間がかかる場合があります．

●コピー料金（税込み）
1ページにつき100円

●発送手数料（判型に関わらず）
1～10ページ：100円，11～30ページ：200円，31～50ページ：300円，51～100ページ：400円，101ページ以上：600円

●送付金額の算出方法
総ページ数×100円＋発送手数料

●入金方法
現金書留か郵便小為替による郵送

●明記事項
雑誌名，年月号，記事タイトル，開始ページ，総ページ数

●宛て先
〒170-8461　東京都豊島区巣鴨1-14-2
CQ出版株式会社　コピー・サービス係
（TEL：03-5395-4211，FAX：03-5395-1642）

■ お問い合わせ先のご案内

●在庫，バックナンバ，年間購読送付先変更に関して
販売部：03-5395-2141

●広告に関して
広告部：03-5395-2133

●雑誌本文に関して
編集部：03-5395-2122

記事内容に関するご質問は，返信用封筒を同封して編集部宛てに郵送してくださるようお願いいたします．筆者に回送してお答えいたします．

Interface

©CQ出版（株）　2004　振替　00100-7-10665
2004年11月号　第30巻　第11号（通巻第329号）
2004年11月1日発行（毎月1日発行）
定価は裏表紙に表示してあります

発行人／増田久喜
編集人／山形孝雄
編集／大野典宏　村上真紀　山口光樹　大竹友美　小林由美子
デザイン・DTP／クニメディア株式会社
表紙デザイン／株式会社プランニング・ロケッツ
本文イラスト／藤波　淳　森　祐子
広告／澤辺　彰　中元正夫　菅原利江

発行所／CQ出版株式会社　〒170-8461　東京都豊島区巣鴨1-14-2
電話／編集部（03）5395-2122　FAX／（03）5395-2127
広告部（03）5395-2133　URL http://www.cqpub.co.jp/interface/
販売部（03）5395-2141　E-mail supportinter@cqpub.co.jp
CQ Publishing Co.,Ltd.／1-14-2 Sugamo, Toshima-ku, Tokyo 170-8461, Japan
印刷／クニメディア株式会社　美和印刷株式会社
製本／星野製本株式会社


日本ABC協会加盟誌
（新聞雑誌部数公査機構）　ISSN0387-9569